PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UM *FRAMEWORK* PARA A ORGANIZAÇÃO DO CONHECIMENTO DE AGENTES DE SOFTWARE

ANA PAULA LEMKE

Orientador: Prof. Dr. Marcelo Blois Ribeiro

Dissertação de Mestrado

Porto Alegre

2007

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ANA PAULA LEMKE

UM *FRAMEWORK* PARA A ORGANIZAÇÃO DO CONHECIMENTO DE AGENTES DE SOFTWARE

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre, pelo programa de Pós Graduação em Ciência da Computação da Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Marcelo Blois Ribeiro

Porto Alegre

2007

Pontifícia Universidade Católica do Rio Grande do Sul

Dados Internacionais de Catalogação na Publicação (CIP)

L554f Lemke, Ana Paula

Um framework para a organização do conhecimento de agentes de software / Ana Paula Lemke. – Porto Alegre, 2007.
131 f.

Diss. (Mestrado em Ciência da Computação) – Fac. de Informática, PUCRS.
Orientação: Prof. Dr. Marcelo Blois Ribeiro.

1. Informática. 2. Sistemas Multiagentes. 3. Framework. 4. Gestão do Conhecimento. I. Ribeiro, Marcelo Blois.

CDD 006.3

Ficha Catalográfica elaborada pelo
Setor de Processamento Técnico da BC-PUCRS

PUCRS

Campus Central

Av. Ipiranga, 6681 - prédio 16 - CEP 90619-900

Porto Alegre - RS - Brasil

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada ***"Um Framework para a Organização do Conhecimento de Agentes de Software"***, apresentada por Ana Paula Lemke, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Sistemas de Informação, aprovada em 24/01/2007 pela Comissão Examinadora:

Prof. Dr. Marcelo Blois Ribeiro – PPGCC/PUCRS
Orientador (a)

Profa. Dra. Vera Lúcia Strube de Lima – PPGCC/PUCRS

Prof. Dr. Ricardo Choren Noya – IME/RJ

Homologada em 08/03/07, conforme Ata No. 005/2007 pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P. 16 – sala 106 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@inf.pucrs.br
www.pucrs.br/facin/pos

AGRADECIMENTOS

Ao meu orientador, Prof. Marcelo Blois Ribeiro, por todos os ensinamentos, críticas e conselhos e, mais do que isso, pela constante presença e paciência nas minhas freqüentes "crises de identidade". Saiba que os desafios lançados colaboraram muito com o meu crescimento, tanto pessoal quanto profissional.

Aos membros da banca, professores Ricardo Choren Noya e Vera Lúcia Strube de Lima, pela aceitação do convite de participação na avaliação deste trabalho. À professora Vera um agradecimento especial por todas as sugestões e críticas dadas ao longo desta pesquisa.

Ao meu namorado Túlio, ao qual havia prometido escrever apenas um "Obrigada", agradeço pelo companheirismo, paciência, amor e carinho dedicados ao longo desta jornada. Amor, agora está "rodando"!

Ao meu irmão Daniel, que muitas vezes sem entender o porquê de minha dedicação incessante a esta pesquisa, suportou o meu stress constante nestes últimos meses e torceu incondicionalmente para que tudo desse certo.

Ao meu padrinho Gilberto e sua família, por todo o apoio e preocupação com meu bem estar. Obrigada também pela acolhida em Porto Alegre.

A três professoras fantásticas que pude conviver durantes esses muitos anos de vida estudantil: Heloisa Krüger, Suzana Tust e Flávia Braga Azambuja. Obrigada pela amizade, exemplos de vida, conselhos e, principalmente, por sempre acreditarem e investirem em mim.

Aos colegas do CDPe e aos membros do grupo de pesquisa ISEG por terem compartilhado muitos momentos agradáveis e também estressantes durantes o período do mestrado. Foi bom ter vocês por perto!

Aos amigos, novos e antigos, pelo companheirismo e motivação. Aos familiares, pela torcida e incentivos para seguir em frente.

Ao Convênio Dell-PUCRS por viabilizar a bolsa de estudos durante os dois anos de mestrado.

Ao Programa de Pós-Graduação em Ciência da Computação e a todos os professores dos quais pude conviver durante estes dois anos.

À Deus, pela sabedoria, persistência e oportunidades que tem posto em minha vida.

E, principalmente, aos meus amados pais, Heroldino e Elaine, por me permitirem sonhar e compartilharem meus sonhos, sempre na torcida para que tudo acabe bem. À vocês dedico tudo, sempre.

"Nosso único patrimônio que realmente faz diferença é o conhecimento".

Peter Drucker

RESUMO

Toda vez que um agente de software adquire conhecimento pela experiência ou pela absorção de novos conceitos é necessário controlar quais são os conhecimentos que o agente domina, ou seja, é necessário gerir seu conhecimento. Sabendo-se que um processo de Gestão de Conhecimento serve como um controlador dos recursos de conhecimento de uma organização, auxiliando a encontrar, organizar e compartilhar o conhecimento, o presente trabalho utilizou-se das partes deste processo para desenvolver um *framework* para a organização do conhecimento de agentes de software, onde o conhecimento é estruturado por meio de ontologias. O *framework* proposto permite aos agentes capturar conhecimento para a execução de suas tarefas e também compartilhar o conhecimento disponível para que ele possa ser reusado por outros agentes do sistema. O entendimento e uso de ontologias e motores de inferência são características fundamentais em aplicações que objetivam utilizar o *framework* proposto, pois, além de estruturar o conhecimento disponível, ontologias são utilizadas na representação do conteúdo de mensagens e dos objetivos dos agentes.

Palavras-chave: Agentes de software. Gestão de Conhecimento. Sistemas Multiagentes. Ontologias.

ABSTRACT

Knowledge acquisition in software agents can be done through experience and explicit concepts' capturing. Both acquisition approaches require knowledge management capabilities in the agents. Knowledge management approaches are usually applied to allow organizations to find, organize, and share knowledge. This work proposes a framework for software agents' knowledge management based on a compilation of those approaches that uses ontology-based knowledge representation. It allows agents to capture knowledge to perform their tasks and to transmit this knowledge to other agents. Ontologies and inference engines are key elements for those who want to use the framework, since ontologies are used not only to represent knowledge, but also to represent agent goals and to encapsulate message content.

Keywords: Software agents. Knowledge Management. Multi-agent systems. Ontologies.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

CIS - *Computational Intelligence Systems*

GC – Gestão de Conhecimento

KO – *Knowledge Object*

OWL – *Ontology Web Language*

RBC – Raciocínio Baseado em Casos

RDF – *Resource Description Framework*

SMA – Sistema Multiagentes

UML - *Unified Modeling Language*

URI – *Universal Resource Identifier*

URL – *Unified Resource Locator*

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de agentes inteligentes de software se confunde com a própria busca de tecnologias inteligentes pelos pesquisadores de Inteligência Artificial. A idéia de se criar um software inteligente, capaz de pensar de forma similar a um ser humano, sempre atraiu inúmeros pesquisadores (FERBER, 1999). A analogia com a capacidade dos seres humanos de pensar, norteou a busca por entidades de software capazes de imitar o comportamento humano (GENESERETH *et al*., 1995).

Um agente de software, segundo Weiss (1999), pode ser definido como um sistema de computador situado em um ambiente e capaz de agir de forma autônoma para atingir um objetivo, onde a autonomia refere-se à capacidade de agir de acordo com sua própria linha de controle.

Para a construção de sistemas complexos, geralmente são utilizados vários agentes que desempenham tarefas voltadas ao alcance de seus objetivos especializados (objetivos locais) e que estão de acordo com os objetivos globais do sistema. Um sistema que possui vários agentes atuando em um ambiente é denominado sistema multiagentes ou SMA (WEISS, 1999). O desenvolvimento de SMAs é uma área em franca expansão devido à sua adaptação à resolução de problemas distribuídos (HOUARI e FAR, 2004).

Existem, atualmente, diversas plataformas disponíveis que auxiliam na criação de aplicações do tipo SMA, como JADE (JADE, 2006), OpenCybele (OpenCybele, 2005) e MadKit (MadKit, 2005). Além dos requisitos como infra-estrutura de comunicação e administração de agentes, que são básicos para uma plataforma que objetiva auxiliar na construção deste tipo de sistema, há outros requisitos também fundamentais. Este é o caso das formas de representação de conhecimento (TODA *et al*., 2001), que permitem a troca estruturada de conhecimento entre os agentes de software.

De acordo com Obitko e Marik (2002), ontologias executam um importante papel no compartilhamento e exploração de conhecimento. As ontologias estabelecem uma terminologia comum usada tanto por humanos quanto por agentes de software para o entendimento dos conceitos de um domínio. Dentre as várias definições para o termo

ontologia existentes na literatura, a mais citada é a oferecida por Gruber: “Uma ontologia é uma especificação explícita de uma conceituação” (GRUBER, 1993).

Sendo um agente uma entidade de software que possui, em tese, a capacidade de aprender (WEISS, 1999) ou, em outras palavras, a capacidade de adquirir conhecimento através de experiência, pela absorção de novos conceitos ou pela reação a mudanças no ambiente, torna-se necessário o uso de mecanismos ou processos para controlar o conhecimento que o agente domina. A Gestão de Conhecimento (GC) refere-se à criação, identificação, integração, recuperação, compartilhamento e utilização do conhecimento dentro do ambiente organizacional. Na prática, GC compreende a identificação e o mapeamento dos recursos intelectuais da organização, gerando conhecimento novo, melhorando a competitividade e tornando uma vasta quantidade de informações acessíveis (MATHI, 2004).

Sabendo-se que um processo de GC serve como um controlador dos recursos de conhecimento de uma organização, auxiliando a encontrar, organizar e compartilhar o conhecimento (LEMKE, 2005) e, considerando-se um SMA como uma organização de agentes (ZHU *et al*., 2004), no presente trabalho será apresentado o desenvolvimento de um *framework* para a organização do conhecimento de agentes de software, com o uso de ontologias e técnicas de GC.

Na literatura foram encontrados vários modelos conceituais de arquiteturas de software para GC corporativa, como as apresentadas em (WEISS, 1999) e (HOUARI e FAR, 2004), que têm como objetivo expandir o alcance e potencializar a velocidade da transferência do conhecimento. Porém, não foram encontradas arquiteturas que utilizam, de forma completa e integrada, um processo de GC com o objetivo de gerir o conhecimento interno do sistema, fato que é a motivação deste trabalho.

Neste contexto, o *framework* proposto tem como principal objetivo permitir à agentes de diferentes arquiteturas internas formalizarem o conhecimento adquirido durante a execução de suas tarefas e também transmitir esse conhecimento para que ele possa ser reusado. A arquitetura conta com procedimentos para o mapeamento de atividades comumente encontradas em um processo de GC, como as atividades criar, organizar, armazenar, distribuir, capturar e aplicar conhecimento.

A representação do conhecimento e o entendimento de ontologias são características fundamentais em aplicações que objetivam usar o *framework* proposto. No *framework*, tanto o formato para representação do conteúdo de mensagens, quanto o

mecanismo de tomada de decisão remetem ao uso de ontologias e motores de inferência. Para a representação de ontologias, optou-se pela linguagem OWL DL (W3C, 2006b), que é baseada em XML e RDF e utiliza lógica descritiva para a explicitação de conhecimento.

Como o *framework* proposto não objetiva servir de base para a construção de SMAs, mas sim agregar uma nova funcionalidade a agentes de software, torna-se necessária a sua integração em plataformas de desenvolvimento de SMAs. Para tanto, deve ser feito o mapeamento entre os elementos contidos no *framework* e os elementos da arquitetura interna dos agentes da plataforma de desenvolvimento selecionada. Neste trabalho, será mostrado o mapeamento sobre o JADE (JADE, 2006), que é uma plataforma para desenvolvimento de SMAs, e sobre o SemantiCore (BLOIS e LUCENA, 2004), que é um *framework* que provê uma definição interna de agente capaz de oferecer aos desenvolvedores uma abstração de alto nível para a construção de SMAs para a *Web* Semântica.

Para demonstrar a viabilidade da arquitetura, depois de sua instanciação sobre o SemantiCore, foi desenvolvida uma aplicação tendo-se como base o exemplo apresentado em (BERNERS-LEE *et al*., 2001). Neste exemplo, há dois agentes, Lucy e Pete, que precisam agendar sessões de fisioterapia para a mãe. Esta aplicação permite a avaliação da arquitetura em termos de sua funcionalidade e abragência com relação à solução do problema proposto.

1.1 Questão de Pesquisa

Considerando-se um SMA como uma organização que tende a se beneficiar com a coleta e classificação do conhecimento disponível entre seus membros e um agente de software como uma entidade que necessita explicitar seu conhecimento para se comunicar de maneira eficiente, parte-se para a análise da adoção de um processo de Gestão de Conhecimento em sociedades de agentes. Neste sentido, emerge a questão de pesquisa deste estudo: "***Como organizar o conhecimento de agentes de software de forma a permitir a recuperação e troca de conhecimento?***".

1.2 Objetivos

Uma vez definida a questão de pesquisa, definiu-se o objetivo geral e os objetivos específicos deste trabalho, os quais são apresentados a seguir.

1.2.1 Objetivo Geral

Propor e aplicar sobre uma plataforma de desenvolvimento de SMAs, um modelo de arquitetura de software que possibilite, através de técnicas de representação de conhecimento e ontologias, a organização do conhecimento disponível em agentes de software.

1.2.2 Objetivos Específicos

- Aprofundar o estudo teórico sobre trabalhos relacionados ao problema abordado.
- Identificar responsabilidades e ações referentes às atividades de um processo de GC em uma arquitetura de agentes.
- Definir o que é um objeto de conhecimento e modelá-lo no escopo do trabalho proposto.
- Propor um modelo de arquitetura de software para Gestão de Conhecimento.
- Mapear e implementar as primitivas do modelo proposto.
- Aplicar o modelo proposto sobre o SemantiCore.
- Projetar e implementar uma aplicação que possibilite o uso do modelo proposto.

1.3 Organização da Dissertação

Este trabalho está dividido em três partes: fundamentação teórica, proposta da arquitetura e caso de validação. O Capítulo 2 apresenta a fundamentação teórica necessária para um bom entendimento do trabalho, onde são esclarecidos aspectos de agentes de software, GC e seus processos associados, *Web* Semântica e ontologias.

No Capítulo 3 são apresentados alguns trabalhos encontrados na literatura que estão relacionados à organização do conhecimento de agentes de software. Durante a apresentação dos trabalhos, vários aspectos do *framework* proposto são salientados, auxiliando na identificação das contribuições desta dissertação.

No Capítulo 4 é descrito o modelo conceitual do *framework* proposto e o seu mapeamento sobre as plataformas JADE e SemantiCore. Além disso, discorre-se sobre a aplicação de um processo de GC em uma arquitetura de agentes. Já no Capítulo 5 são apresentadas as diretrizes de implementação do *framework*, indicando as técnicas utilizadas para a construção de seus elementos.

O Capítulo 6 apresenta um exemplo de validação que demonstra a implementação de um SMA. Para a apresentação do sistema, trechos de código são descritos, com explicações contextualizadas sobre como usar o *framework* e também sobre como definir os objetos de conhecimento. Por fim, no Capítulo 7 são apresentadas as conclusões e os trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Para um bom entendimento da arquitetura proposta e dos trabalhos relacionados é necessária a compreensão dos fundamentos teóricos e das tecnologias básicas nos quais a arquitetura se baseia. Assim, nas seções 2.1 e 2.2 são apresentadas características de agentes de software e SMAs. Como a arquitetura proposta deverá ser integrada a plataformas de desenvolvimento de SMAs, na Seção 2.3 são descritas quatro plataformas disponíveis na literatura. Já na Seção 2.4, é esclarecido o conceito de GC e é apresentada também, uma síntese de processos de GC encontrados na literatura. Ainda na Seção 2.4, fala-se do uso da técnica de raciocínio baseado em casos em GC. O uso da *Web* como um meio por onde agentes navegam e fazem requisições, trocando informações com anotações semânticas, é o tema da seção 2.5, que trata de *Web* Semântica e ontologias.

2.1 Agentes de Software

Existe, atualmente, um grande debate sobre como conceituar exatamente um agente de software. Entre as muitas definições usadas na literatura, cita-se a dada por Wooldridge em (WOOLDRIDGE, 2002):

> *"Um agente é um sistema de computador encapsulado que está situado em um ambiente e que é capaz de agir de forma flexível e autônoma neste ambiente a fim de satisfazer seus objetivos"*.

Para entender melhor o que é um agente, é necessário entender primeiro quais são as suas características. Muitas definições são usadas na literatura, mas algumas características permanecem em todas elas. São estas características que serão utilizadas para conceituar um agente de software neste trabalho. Assim, um agente é uma entidade de software que:

- É autônoma – isto significa que, uma vez criado, um agente age de forma autônoma com relação às outras entidades de um sistema. Isto não quer dizer que um agente não se comunique ou coopere com outras entidades,

mas que o seu fluxo de execução é independente e, mais importante, que o agente tem autonomia de atuação (onde atuação não quer dizer apenas fluxo de execução independente, mas existência de mecanismos de dedução, associação e indução).

- Atua em um ambiente – o conceito de ambiente relaciona-se fortemente a existência de vários agentes atuando em um sistema e também a capacidade do agente de perceber o que acontece ao seu redor.
- É inteligente – a inteligência deve ser entendida neste contexto como a capacidade de atuar segundo objetivos e de acordo com o conhecimento que o agente possui do seu ambiente e das suas ações.
- Possui um modelo limitado do mundo – o seu escopo limitado deve-se à complexidade das entidades que atuam no ambiente e aos objetivos específicos de cada agente. Com isso, um agente só possui o modelo de mundo necessário ao desempenho de suas tarefas e ao alcance de seus objetivos.

De maneira resumida, um agente é uma entidade de software que, a partir de informações sentidas no ambiente, captadas através da interação direta com outros agentes de software ou humanos, ou geradas a partir dos mecanismos dedutivos internos ao agente, atua em um ambiente buscando o alcance de seus objetivos (RIBEIRO, 2002).

2.2 Sistemas Multiagentes

Lo *et al.* (2002) definem um SMA como um "grupo de agentes que interagem entre si para resolver um problema ou cumprir uma tarefa numa plataforma baseada em agentes". Algumas características de SMAs foram sumarizadas por Zhu e co-autores em (ZHU *et al.*, 2003). Segundo estes autores, em SMAs: (i) cada agente tem uma habilidade parcial para resolver um determinado problema; (ii) não há um controle global do sistema; (iii) dados e conhecimento para solucionar um problema são descentralizados; e (iv) a computação é assíncrona.

Ribeiro (2002) dá a seguinte definição para SMA:

> *"Um SMA é um sistema formado por diversos agentes que mantém alguma relação entre as suas ações e que atuam em um ambiente."*

Nesta definição, o conceito de ambiente fica bastante evidente como sendo uma parte de um SMA. O ambiente delimita o escopo de atuação dos agentes, servindo como base de informações para os sensores e como canal de saída das ações dos agentes. Outra questão que é salientada nesta definição é a relação que deve existir entre as diferentes ações dos agentes de um SMA. Esta restrição indica que as ações dos agentes devem estar associadas de alguma forma e que esta associação deve servir aos objetivos individuais dos agentes e ao objetivo de todo o sistema ou sociedade de agentes (RIBEIRO, 2002).

Considerando que os agentes em um espaço aberto são autônomos, ou seja, que o agente pode perceber o ambiente e agir baseado nos resultados percebidos, é razoável considerar-se um SMA como uma sociedade (ZHU *et al.*, 2004), onde os agentes vivem e executam tarefas. O entendimento de um SMA como uma sociedade ou organização de agentes será referenciado no decorrer deste trabalho.

2.3 Desenvolvimento de Sistemas Multiagentes

Como as técnicas de Engenharia de Software possuem limitações quanto à representação de requisitos específicos de SMAs (WOOLDRIDGE *et al*., 1999), vêm sendo propostas algumas arquiteturas que incorporam conceitos de agência nativos em seus modelos. Estas arquiteturas tiveram origem em esforços de consórcios de instituições de pesquisa e em empresas de grande porte, que procuram utilizar SMAs como solução de problemas distribuídos complexos (OMG, 2000). Embora exista um avanço na criação de plataformas para a implementação de SMAs, não existe nenhuma plataforma que seja considerada como padrão para todos os domínios de aplicações existentes.

Para que uma plataforma de implementação de SMAs seja completa é preciso tanto o suporte ao desenvolvimento da parte interna dos agentes quanto o suporte à criação da infra-estrutura de atuação dos agentes em sua organização (parte externa). Dentre as

plataformas disponíveis para a implementação de SMAs, como o Madkit, o JADE, o OpenCybele e o SemantiCore[1], são poucas as que oferecem suporte total à sua confecção, como será mostrado a seguir, através da descrição das plataformas citadas.

2.3.1 Plataformas de desenvolvimento de SMAs

2.3.1.1 Madkit

O kit de ferramentas MadKit - *a Multi-Agent Development Kit* - surgiu da necessidade de fornecer uma plataforma genérica de SMAs, altamente adaptável e escalável (FERBER *et al.*, 2005). O objetivo era construir uma infra-estrutura para modelos de agentes variados e fazer os serviços básicos inteiramente extensíveis e substituíveis (GUTKNECHT e FERBER, 2000).

Segundo Ferber (2005), MadKit é uma plataforma multiagentes para o desenvolvimento e execução de aplicações baseadas em um paradigma orientado a organização. A plataforma MadKit é construída de acordo com o modelo *Aalaadin* (GUTKNECHT e FERBER, 2000), ilustrado na Figura 2.1, onde:

- Agente: classe que define o ciclo de vida abstrato básico. Um agente é especificado como uma entidade de comunicação ativa que executa papéis dentro do grupo (a arquitetura interna dos agentes não é explicitada no modelo). A definição de agente é intencionalmente genérica para permitir que os desenvolvedores adotem a definição mais propícia para a sua aplicação.
- Grupos: são definidos como conjuntos atômicos de agentes agregados. Cada agente é parte de um ou mais grupos.

[1] O SemantiCore será definido, ao longo do trabalho, como uma plataforma para desenvolvimento de SMAs por auxiliar no desenvolvimento de aplicações baseadas em agentes na *Web* Semântica.

- Papel: é uma representação abstrata de uma função, serviço ou identificação do agente dentro de um grupo. Cada agente pode ter múltiplos papéis e cada papel executado por um agente é local no grupo.

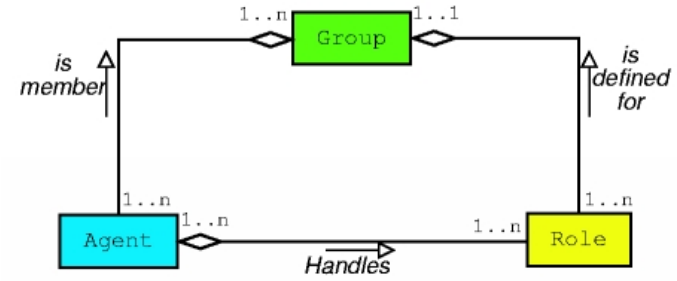


Figura 2.1: Modelo principal do MadKit (GUTKNECHT e FERBER, 2000)

Em adição a esses três conceitos elementares, Madkit tem três princípios para o desenvolvimento (GUTKNECHT e FERBER, 2000; FEBER *et al.*, 2005):

- Arquitetura de micro-*kernel*: é um pequeno e otimizado *kernel* do agente que tem como função controlar grupos e agentes locais, gerenciar o ciclo de vida do agente e transmitir mensagens locais.
- "Agentificação" de serviços: no MadKit, todos os serviços, exceto os fornecidos pelo *kernel*, são executados por agentes.
- Modelo de componente para interface gráfica: a interface gráfica é baseada na especificação do Java *Bean*. Cada agente é o único responsável por sua interface gráfica e seus aspectos, como *rendering*, eventos, processamento, ações, entre outros.

De maneira simplificada, os serviços do sistema são oferecidos por um agente especial, o "*KernelAgent*", e pelo grupo "*System*". Os agentes são separados em grupos e existe um agente especial com o papel de facilitador ("*Broker*"), que localiza os agentes na sociedade. A comunicação ocorre por um sistema de mensagens onde podem ser enviadas tanto mensagens síncronas quanto assíncronas (FERBER *et al.*, 2005).

2.3.1.2 JADE

JADE (*Java Agent DEvelopment Framework*) é um *framework* de desenvolvimento de software totalmente implementado na linguagem Java. Tem como objetivo suportar o desenvolvimento de aplicações de SMAs através de um *middleware*[2] (que segue as especificações da FIPA - *Foundation for Intelligent Physical Agents*) e de um conjunto de ferramentas gráficas que suportam as fases de desenvolvimento e verificação (JADE, 2006).

JADE permite o desenvolvimento de sistemas capazes de trabalhar de uma maneira pró-ativa (de acordo com regras pré-definidas), de se comunicar e negociar diretamente com outras partes do sistema e de se coordenar a fim de solucionar problemas complexos de maneira distribuída. (BELLIFEMINE *et al.*, 2005; JADE, 2006).

Dentre a lista de características do JADE, apresentada por Bellifemine e co-autores em (BELLIFEMINE *et al.*, 2005), destacam-se:

- Plataforma de agentes distribuída: o JADE pode ser dividido em vários *hosts* ou máquinas, desde que eles possam ser conectados via RMI (*Remote Method Invocation*). Apenas uma aplicação Java e uma Java *Virtual Machine* é executada em cada *host.* Os agentes são implementados como *threads* Java e são inseridos dentro de repositórios de agentes chamados de *Agent Containers*, que provêem todo o suporte para a execução do agente.
- Interface gráfica: interface visual que permite gerenciar vários agentes e repositórios de agentes, inclusive remotamente.
- Ferramentas de depuração: ferramentas que ajudam no desenvolvimento e na depuração de aplicações multiagentes baseadas em JADE.
- Transporte de mensagens: transporte de mensagens no formato FIPA-ACL dentro da mesma plataforma de agentes.

---

[2] *Middleware* pode ser definido como uma camada de software que concentra funcionalidades tradicionalmente dispersas entre aplicações.

- Ambiente de agentes complacente a FIPA: o JADE contém um sistema gerenciador de agentes (*Agent Management System*), um facilitador de diretórios (*Directory Facilitator*) e um canal de comunicação dos agentes (*Agent Communication Channel*). Todos estes componentes são automaticamente carregados quando o ambiente é iniciado.

2.3.1.3 OpenCybele

OpenCybele é um ambiente construído em Java para controle e execução de agentes baseados em eventos, ou seja, os agentes executam de acordo com os eventos percebidos do ambiente. É uma releitura de código aberto do Cybele™, desenvolvida pelo *Intelligent Automation Incorporated*[3] (IAI). Com a sua arquitetura de camadas de serviços são promovidas diversas capacidades *plug-and-play* de serviços, tais como tratamento de erros, gerenciamento de *threads*, tratamento de eventos, comunicação e migração.

No OpenCybele, os serviços de agentes são categorizados dentro de três tipos de camadas: básica, fundamental e suplementar (as camadas estão representadas graficamente na Figura 2.2). Essa distribuição é baseada na relevância do serviço para o agente que está executando na plataforma (OpenCybele, 2005).

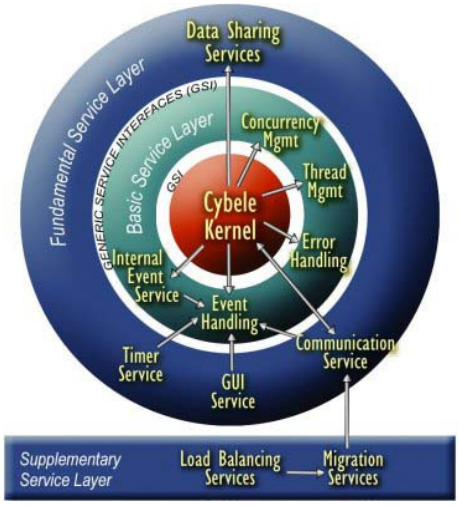


Figura 2.2: Arquitura de camadas de serviços do OpenCybele (OpenCybele, 2005)

[3] www.i-a-i.com

As aplicações de agentes desenvolvidas com o OpenCybele interagem somente com a infra-estrutura do *Cybele kernel*, que depende, por sua vez, das execuções subjacentes dos serviços de infra-estrutura para executar de acordo com o paradigma baseado em agentes do *Cybele*.

2.3.1.4 SemantiCore

O SemantiCore é estruturado como um *framework* para omitir ligações específicas da plataforma e para prover primitivas para a criação de aplicações organizadas em um conjunto de agentes que realizam suas tarefas no ambiente *Web* (BLOIS e LUCENA, 2004). O SemantiCore, apresentado inicialmente em 2004 (BLOIS e LUCENA 2004), surgiu a partir de uma extensão na arquitetura *Web Life* (Ribeiro, 2002) e atualmente se encontra disponível na versão 2006 – SemantiCore 2006 (ESCOBAR *et al.*, 2006).

O *framework* SemantiCore é dividido em dois modelos: o modelo do agente (*SemanticAgent*), responsável pelas definições internas dos agentes, e o modelo do domínio semântico, responsável pela definição da composição do domínio e suas entidades administrativas. Os dois modelos dispõem de pontos de flexibilidade (*hotspots*) permitindo aos desenvolvedores associar diferentes padrões, protocolos e tecnologias.

O modelo do agente possui uma estrutura orientada a componentes, onde cada componente contribui para uma parte essencial do funcionamento do agente, agregando todos os aspectos necessários a sua implementação. Com a retirada de um ou mais componentes não relacionados ao desempenho das tarefas do agente é possível simplificar a sua arquitetura. São quatro os componentes básicos do agente:

- Sensorial: permite que o agente recupere objetos a partir do ambiente. O componente sensorial armazena os diversos sensores definidos pelo desenvolvedor (cada sensor captura um tipo diferente de objeto do ambiente) e também verifica se algum destes sensores deve ser ativado pelo recebimento de alguma mensagem do ambiente. Se um ou mais sensores forem ativados, os objetos são enviados para os outros componentes para processamento. Um sensor OWL (*OWLSensor*) é um

tipo especial de sensor já definido na plataforma que captura objetos em OWL no ambiente.

- Decisório: encapsula o mecanismo de tomada de decisão do agente. O mecanismo decisório presente no componente é um dos pontos de flexibilidade do *framework* (*hotspot*). Embora o mecanismo decisório seja um *hotspot*, o SemantiCore possui uma integração nativa com o Jena (JENA, 2006), possibilitando o uso de máquinas de inferência neste componente. Para que a saída gerada pelo componente decisório possa ser entendida, ela deve ser uma instância de uma ação (*Action*). As ações mapeiam todos os possíveis comandos que um agente deve entender para trabalhar de forma apropriada. O desenvolvedor pode definir suas próprias ações através da extensão da classe *Action* (*hotspot*) presente no *framework*.
- Executor: contém os planos de ação que serão executados pelo agente e pode trabalhar com o mecanismo de *workflow*.
- Efetuador: recebe dados dos outros componentes e encapsula estes em objetos semânticos para serem transmitidos no ambiente. Toda publicação de um objeto semântico no ambiente requer um efetuador apropriado no agente.

Para que um agente possa atuar, é necessário que ele esteja situado em um ambiente. No SemantiCore, este ambiente é denominado domínio semântico. Um domínio semântico requer um domínio *Web* para operar. Como ilustrado na Figura 2.3, cada domínio semântico é composto por algumas entidades administrativas, como o **Controlador de Domínio** (*Domain Controller*) e o **Gerente de Ambiente** (*Environment Manager*). O Controlador de Domínio é responsável por registrar os agentes no ambiente, pela recepção de agentes móveis vindos de outros domínios e também pela manutenção e execução de aspectos relacionados a segurança. O Gerente de Ambiente representa uma ponte entre o domínio semântico do SemantiCore e os domínios *Web* convencionais.

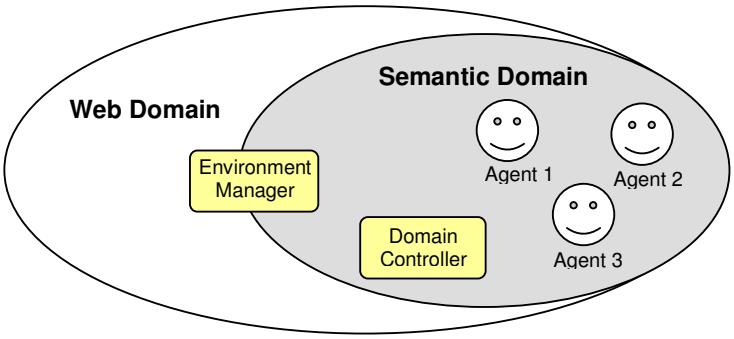


Figura 2.3: Modelo de domínio do SemantiCore (ESCOBAR *et al*., 2006)

2.4 Gestão de Conhecimento

Para permanecerem competitivas, as organizações devem ser capazes de criar, encontrar, capturar e compartilhar o conhecimento de forma eficiente e eficaz. Isto requer, cada vez mais, tornar explícito o conhecimento organizacional, gravando-o de forma a facilitar sua distribuição e reusabilidade. As organizações têm de responder rápida e eficazmente às constantes mudanças e a Gestão de Conhecimento apareceu como resposta a este desafio.

Gestão de Conhecimento (GC) pode ser entendida como o gerenciamento explícito e sistemático do conhecimento vital e seus processos associados de criação, organização, difusão, uso e exploração. De forma mais simplificada, GC é o processo de converter conhecimento vindo de fontes disponíveis para a organização e conectar pessoas com este conhecimento (DEVEDZIC, 2002).

Para Stader e Macintosh (1999), GC pode ser definida como a identificação e análise dos recursos de conhecimento disponíveis e requeridos e também dos recursos relatados em processos. Esta definição de GC implica que é necessário para a organização:

- Ser capaz de identificar e representar seus recursos de conhecimento.
- Compartilhar e reusar estes recursos de conhecimento para diferentes aplicações e usuários, o que implica ter o conhecimento disponível onde ele é necessário dentro da organização.

- Criar uma cultura que encoraje o compartilhamento e reuso de conhecimento.

Da mesma forma que nas organizações empresariais, uma sociedade de agentes é constituída por indivíduos (agentes) que necessitam explicitar e compartilhar o conhecimento adquirido durante a execução de suas tarefas, criando assim uma memória organizacional. Deste modo, um SMA é visto como uma organização que tende a se beneficiar com a coleta e classificação do conhecimento disponível entre seus agentes.

Neste contexto, o desafio das organizações, de software ou não, é aprender a converter o conhecimento de seus colaboradores em conhecimento organizacional, ou seja, explicitar o conhecimento dentro da organização. É necessário o desenvolvimento de métodos que auxiliem as organizações a encontrar, selecionar, organizar, disseminar e transferir informações importantes (MATHI, 2004).

2.4.1 Processo de Gestão de Conhecimento

Um processo de Gestão de Conhecimento (*Knowledge Management Process*) serve como um controlador dos recursos de conhecimento da organização, auxiliando a encontrar, organizar e compartilhar o conhecimento. Ele é representado por uma seqüência de atividades através da qual o conhecimento organizacional é conduzido, da captura até o compartilhamento propriamente dito. O processo de GC deve ser encaixado em todo o projeto, processo, comunidade ou rede da organização.

Com o objetivo de estabelecer um processo de GC para ser usado como guia no decorrer da pesquisa, em (LEMKE, 2005) foi feita uma análise de propostas de processos de GC encontradas na literatura (KUCZA, 2001; ZACK, 1999; LEE e LEE, 2003; NISSEN, 1999; DESPRES e CHAUVEL, 1999; GARTNER GROUP, 1998; DAVENPORT e PRUSAK, 1998), cujos resultados estão apresentados na Tabela 2.1.

De acordo com a Tabela 2.1, a maioria das sete propostas começa com a atividade “criar” ou “capturar”; apenas o modelo de Kucza inicia com a identificação do conhecimento - atividade que não aparece em nenhuma fase dos outros modelos. A segunda fase pertence à organização, onde o conhecimento é mapeado, refinado e encapsulado –

novamente o modelo de Kucza diferencia-se nesta fase, pois já parte para o compartilhamento do conhecimento. Na fase três, são usados termos diferentes nas propostas, mas todas, exceto a de Lee e Lee, indicam algum mecanismo de armazenamento/formalização do conhecimento. Também, embora as propostas tragam uma nomenclatura diferente na fase quatro, quase todas implementam o compartilhamento do conhecimento na organização. Quatro das sete propostas incluem uma quinta fase para apresentação, aplicação ou uso/reuso de conhecimento. Apenas a proposta de Despres e Chauvel inclui uma sexta fase que representa a evolução do conhecimento.

Tabela 2.1: Atividades e fluxo de atividades de um processo de GC

Modelo	Fase 1	Fase 2	Fase 3	Fase 4	Fase 5	Fase 6
Kucza	Identificar	Compartilhar	Coletar ou armazenar conhecimento	Atualizar		
Zack	Adquirir	Refinar	Armazenar/ Recuperar	Distribuir	Apresentar	
Lee e Lee	Coletar	Relacionar	Criar	Distribuir		
Nissen	Capturar	Organizar	Formalizar	Distribuir	Aplicar	
Despres e Chauvel	Criar	Mapear	Armazenar	Compartilhar / Transferir	Reusar	Evoluir
Gartner Group	Criar	Organizar	Capturar	Acessar	Usar	
Davenport & Prusak	Gerar		Codificar	Transferir		
Síntese	**Capturar / Criar**	**Organizar**	**Armazenar**	**Distribuir**	**Aplicar**	

A última linha da Tabela 2.1 (denominada “síntese”) apresenta o termo que aparece com maior freqüência em cada uma das fases (muitas vezes aparecem sinônimos do termo), de acordo com as propostas de processo analisadas. Este será o fluxo de atividades que representará um processo de GC no decorrer deste trabalho.

2.4.2 Raciocínio baseado em casos

Raciocínio baseado em casos (RBC) é uma técnica de IA que modela aspectos da cognição humana para resolver problemas especializados. Sistemas de RBC imitam o ato humano de recordar um episódio prévio para resolver um determinado problema devido a

forte semelhança entre eles. No processo de recordar uma situação semelhante quando comparado a uma nova, sistemas de RBC simulam o raciocínio analógico (RIESBECK e SCHANK, 1989).

O ato de relembrar um episódio anterior é simulado em um sistema de RBC por meio da comparação de um novo problema com um conjunto de casos do mesmo tipo (base de casos). A comparação entre os casos é feita através da avaliação de similaridade entre o novo episódio com os já contidos na base de casos, sendo que somente os casos mais similares são recuperados. Para completar o ato de relembrar, há uma fase de seleção onde é determinado qual o caso mais útil.

São quatro as etapas principais no desenvolvimento de um sistema de RBC: **recuperar**, **reutilizar**, **revisar** e **reter**. Autores como Aamodt e Plaza (1994) referem-se a estas etapas como o ciclo do RBC.

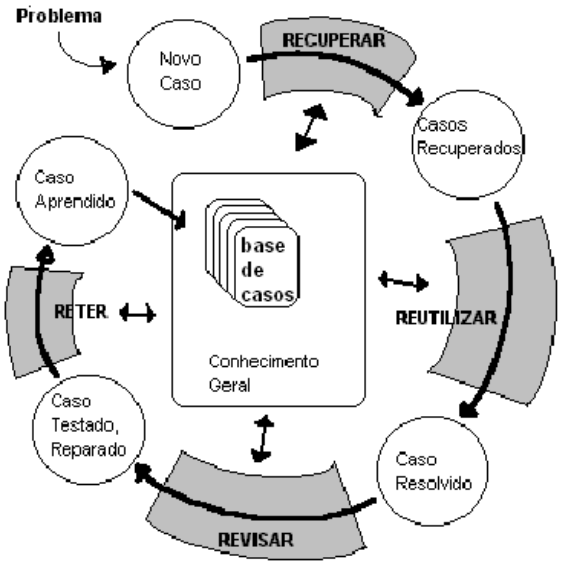


Figura 2.4: O ciclo do RBC (AAMODT e PLAZA, 1994)

A Figura 2.4 ilustra as quatro principais etapas de um sistema de RBC citadas. Essas etapas podem ser descritas da seguinte maneira (AAMODT e PLAZA, 1994):

- Recuperar: busca de um ou mais casos de acordo com as similaridades entre a situação atual (caso de entrada) e cada um dos casos da base (casos candidatos).

- Reutilizar: utilização da informação e do conhecimento contidos nos casos recuperados (pode ser apenas um) para resolver o caso de entrada.
- Revisar: avaliação da solução proposta.
- Reter: armazenamento do caso de entrada juntamente com sua solução revisada na base de casos. Esta etapa representa a característica de aprendizagem de um sistema RBC.

Na literatura, RBC tem sido apontado como um método apropriado para suportar GC e seus processos (VALENTE, 2004; WEBER e KAPLAN, 2003). As principais razões para esta indicação são: (1) as similaridades existentes entre os processos de RBC e os processos de GC e; (2) similaridades heurísticas, visto que a metodologia utilizada em RBC é baseada na forma do raciocínio humano, que por sua vez é freqüentemente utilizada em processos de GC.

2.5 Web Semântica e Ontologias

O conceito de *Web* Semântica foi introduzido no meio científico em meados de 2001 por Tim Berners-Lee, James Hendler e Ora Lassila (BERNERS-LEE *et al*., 2001). A *Web* Semântica é uma iniciativa que busca a definição de formatos padrão para exprimir informações em uma forma processável pela máquina (SEMANTIC WEB, 1998). A idéia básica é criar formas de explicitar o relacionamento de conceitos de vários domínios de conhecimento, permitindo que máquinas possam trabalhar sobre estes conceitos, relacionando-os e inferindo novos conceitos.

A distribuição de tarefas é essencial para a *Web* Semântica, o que remete ao estudo de sistemas distribuídos ou agentes de software. Na *Web* Semântica, agentes de software são usados como entidades capazes de consumir automaticamente conteúdos publicados. Na literatura são encontrados diversos trabalhos que citam a contribuição e uso de agentes de software na *Web* Semântica, como (HENDLER, 2001; LI, 2002), e trabalhos que visam apoiar o desenvolvimento de aplicações deste tipo, como (GUO *et al.*, 2005; BLOIS e LUCENA, 2004).

Na *Web* Semântica, é através do uso de ontologias que o domínio é representado de forma semântica. Ontologias podem ser descritas como a representação explícita de um domínio com seus conceitos, propriedades, atributos e restrições (FREITAS *et al.*, 2005). A motivação para o uso de ontologias é a representação semântica de um domínio específico.

As ontologias não apresentam sempre a mesma estrutura, mas existem características e componentes básicos comuns em grande parte delas. Os componentes básicos de uma ontologia são as classes (organizadas em uma taxonomia), as relações (representam o tipo de interação entre os conceitos de um domínio), os axiomas (usados para modelar sentenças sempre verdadeiras) e as instâncias (utilizadas para representar elementos específicos, ou seja, os próprios dados) (GRUBER, 1996; NOY e MCGUINNESS, 2001).

O uso de ontologias está também associado a GC (O'LEARY, 1998). Segundo Maedche (2002), o vínculo entre GC e ontologias está na estruturação dos processos do conhecimento, que estão interconectados de forma bastante flexível. Usando-se ontologias, a informação passa a ser armazenada como uma informação semântica integrada, gerando aos usuários visões que tornam fácil o acesso ao conhecimento. Já em Weber e Kaplan (2003), tem-se que a associação entre GC e ontologias se dá devido ao propósito de GC, que compreende suportar a representação de conhecimento além de seu compartilhamento e aquisição. Também, segundo estes autores, o uso de ontologias pode auxiliar na resolução de dois desafios encontrados no desenvolvimento de sistemas baseados em conhecimento: o alto custo de aquisição de conhecimento e a falta de conhecimento de senso comum.

2.5.1 Linguagens para descrição de ontologias

Ao decidir-se pela utilização de ontologias, alguns aspectos devem ser observados. Um desses aspectos refere-se à criação da ontologia. Sempre que possível, deve-se utilizar uma ontologia já existente para um determinado domínio. Existem, na *Web*, repositórios especializados em armazenar ontologias definidas por grupos de especialistas. Uma vez que já exista uma ontologia semelhante na área de interesse, o trabalho a ser realizado passa a ser estender a ontologia, acrescentando os conceitos e relações pertinentes

ao domínio em questão. O uso de ferramentas para a criação de ontologias também deve ser observado.

Outro aspecto diz respeito ao modo de representação da ontologia. Existem, atualmente, várias formas de representar as ontologias por meio de linguagem de marcação. Essas linguagens objetivam, principalmente, representar as informações através da descrição formal de um conjunto de termos sobre um domínio específico. De acordo com Antoniou e Harmelen (2004), as mais importantes linguagens de descrição de ontologias disponíveis são:

- XML - *eXtensible Markup Language*: provê uma sintaxe superficial para documentos estruturados e tem restrições quanto a representação do significado dos documentos.
- RDF - *Resource Description Framework*: é um modelo de dados com semântica simples para descrever objetos (recursos) e relacionamentos entre eles.
- RDF *Schema*: é uma linguagem de descrição de vocabulários para descrever propriedades e classes de recursos RDF.
- OWL - *Ontology Web Language*: utiliza lógica descritiva para a explicitação de conhecimento. Permite descrever propriedades e classes, assim como relações entre as classes (como *disjointness*), cardinalidade (como "exatamente um"), igualdade, características das propriedades (como simetria) e classes enumeráveis.

Para os objetivos deste trabalho, serão verificadas mais detalhadamente duas das linguagens citadas: RDF e OWL (sendo este último recomendado pela W3C como linguagem para manipulação de ontologias) (W3C, 2006b).

2.5.1.1 RDF

O RDF é um modelo de dados básico, do tipo entidade-relacionamento, para escrever declarações sobre objetos *Web* (recursos). Seus conceitos fundamentais, conforme ilustrado na Figura 2.5, são (ANTONIOU e HARMELEN, 2004):

- Recurso (*Resource*): qualquer coisa sobre o que se possa falar. Todo recurso tem uma URI (*Universal Resource Identifier*) associada. Uma URI pode ser uma URL (*Unified Resource Locator* ou endereço *Web*) ou qualquer outro tipo de identificador único.
- Propriedade (*Property*): tipo especial de recurso; as propriedades descrevem relações entre os recursos e também são identificadas por URIs.
- Declaração (*Statement*): é uma tripla do tipo (sujeito, predicado, objeto) consistindo de um recurso, uma propriedade e um valor. Valores podem ser recursos ou literais.

A Figura 2.6 mostra um exemplo de grafo e seu código RDF correspondente. O código inicia com a declaração do prólogo (linha 1) e nas linhas 2, 3 e 4 são definidos os *namespaces*[4]. Uma vez especificados os *namespaces*, pode-se utilizar seus descritores de forma não-ambígua ao longo do documento (na linha 6, por exemplo, é utilizado o prefixo "ex" para descrever informações do editor referentes ao *namespace* definido na linha 4).

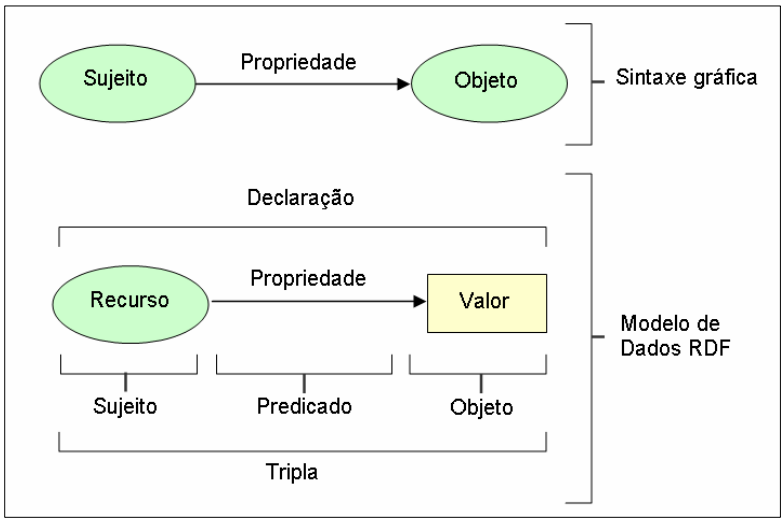


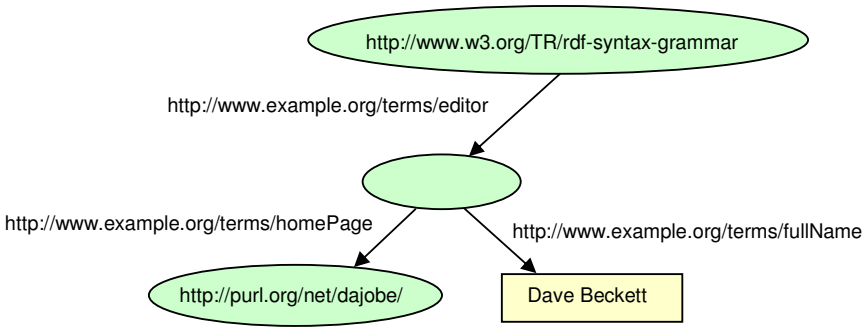
Figura 2.5: Modelo de dados RDF

RDF é genérico e, portanto, não faz associações sobre nenhum domínio de aplicação em particular. Para especificar a semântica do domínio usa-se RDF *Schema* ou

[4] Um *namespace* define um vocabulário controlado que identifica um conjunto de conceitos de forma única para que não haja ambigüidade na sua interpretação.

RDFS (ANTONIOU e HARMELEN, 2004). Segundo Daconta *et al.* (2003), RDFS é um conjunto simples de classes e propriedades RDF para definição de novos vocabulários RDF. Os elementos básicos de um RDFS são:

- Classes (*Class*): definem tipos de objetos. Assim como em linguagens orientadas a objetos, uma classe é definida como um grupo de coisas com características comuns. Em RDFS é possível estabelecer relações de hierarquia (através da propriedade "*is a subclass of*") e herança entre as classes.
- Propriedades (*property*): são definidas globalmente, ou seja, não são encapsuladas como atributos nas definições das classes (Antoniou e Harmelen, 2004). Existem propriedades para definir relações (como "*rdfs:subClassOf*") e para definir restrições de propriedades (como "*rdfs:range*" e "*rdfs:domain*").




```
1 <?xml version="1.0"?>
2 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3    xmlns:dc="http://purl.org/dc/elements/1.1/"
4    xmlns:ex="http://example.org/stuff/1.0/">
5  <rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar">
6    <ex:editor>
7      <rdf:Description>
8        <ex:homePage>
9          <rdf:Description rdf:about="http://purl.org/net/dajobe/">
10         </rdf:Description>
11       </ex:homePage>
12       <ex:fullName>Dave Beckett</ex:fullName>
13     </rdf:Description>
14   </ex:editor>
15  </rdf:Description>
16 </rdf:RDF>
```

Figura 2.6: Exemplo de código RDF (W3C, 2006a)

RDF e RDFS permitem a representação de alguns conhecimentos ontológicos, mas possuem várias limitações. De acordo com Antoniou e Harmelen (2004), algumas das limitações dessas linguagens são:

- Propriedades de escopo local: "*rdfs:range*" define o alcance de uma propriedade para todas as classes. Em RDFS não é possível declarar restrições de alcance para aplicar somente em algumas classes.
- Restrições de cardinalidade: restrições do tipo "apenas um" ou "exatamente dois" não podem ser representadas em RDFS.
- Características especiais de propriedades: em RDFS também não é possível indicar se uma propriedade é transitiva, única ou inversa de outra propriedade, por exemplo.

2.5.1.2 OWL

OWL é uma linguagem de marcação baseada em XML e RDF para a definição e a instanciação de ontologias processáveis computacionalmente. Pode ser usada para representar o significado de termos em um vocabulário e as relações entre esses termos (GEROIMENKO, 2003).

A linguagem OWL se subdivide em 3 sublinguagens (POWERS, 2003):

- OWL *Lite*: permite fazer a classificação hierárquica dos termos relacionados ao domínio e restrições simples.
- OWL DL: estende a linguagem OWL *Lite* e permite fazer restrições de cardinalidade diferentes de 0 ou 1. Em OWL DL todas as conclusões são garantidamente computacionais e todas as computações terminam em um tempo finito.
- OWL *Full*: suporte total para a liberdade máxima do RDF, sem garantias computacionais e com possibilidade de processamento em tempo infinito.

Um documento OWL é composto de indivíduos, classes e propriedades, onde (PROTÉGÉ, 2005):

- Indivíduos: são também conhecidos como instâncias; representam objetos de um domínio específico.
- Classes: são conjuntos que contêm indivíduos com características semelhantes. São descritas através de formalismos que retratam precisamente os requisitos para uma determinada sociedade de classes.
- Propriedades: são relações binárias entre indivíduos. Em OWL, propriedades podem ter propriedades inversas (por exemplo, a propriedade "tem_partes" é inversa da propriedade "é_sub_Parte_De") além de possuir características como transitividade e simetria. Há dois tipos de propriedades:
    - De objetos (*Object properties*): relacionam um indivíduo com outro indivíduo.
    - De tipo de dados (*Datatype properties*): relacionam um indivíduo a um valor XML *Schema Datatype* (*string*, *integer*, entre outros) ou a um literal RDF.

A Figura 2.7 mostra um fragmento de código OWL onde é definida a estrutura de uma classe chamada *Wine*. Nela podemos identificar uma propriedade chamada *madeFromGrape* que tem restrição de cardinalidade mínima igual a 1 (linhas 4 a 8).

```
1  <owl:Class rdf:ID="Wine">
2    <rdfs:subClassOf rdf:resource="&food;PotableLiquid"/>
3    <rdfs:subClassOf>
4      <owl:Restriction>
5        <owl:onProperty rdf:resource="#madeFromGrape"/>
6        <owl:minCardinality
7 rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
8      </owl:Restriction>
9    </rdfs:subClassOf>
10   ...
11 </owl:Class>
```

Figura 2.7: Exemplo de código OWL (W3C, 2006b)

2.5.2 Inferência em ontologias

A partir de uma ontologia é possível recuperar conhecimento, de acordo com sua semântica, através do uso de um motor de inferência. Por meio das regras de inferência, podem-se derivar novos fatos baseados em fatos existentes, ou seja, inferem-se novas informações.

Uma das ferramentas existentes para o auxílio computacional ao processo de inferência é o Jena (JENA, 2006), projeto *open-source* desenvolvido pelo *HP Labs Semantic Web Programme*, que é um *framework* para construção de aplicações voltadas à *Web* Semântica que fornece um ambiente de programação para OWL (e outras linguagens) e inclui um motor de inferência baseado em regras. Por meio do Jena é possível não só manipular, consultar e persistir arquivos OWL, mas também criar novos motores de inferência ou mesmo estender os motores já existentes.

2.6 Considerações sobre o capítulo

As seções anteriores apresentaram as tecnologias básicas utilizadas na construção do *framework* proposto. Inicialmente, as definições de agente e SMA apresentadas esclarecem as características fundamentais destas entidades, fornecendo indícios para a criação do modelo de arquitetura. O fato de o agente atuar de acordo com objetivos, por exemplo, indica que o agente está relacionado a um elemento "objetivo" e que este elemento, por sua vez, deve estar relacionado a um conjunto de ações que, quando executadas, permitem que o objetivo seja atingido. Já o estudo das plataformas disponíveis para desenvolvimento de SMAs, além de possibilitar a verificação da ausência de suporte nativo à organização do conhecimento dos agentes, permitiu o levantamento de características básicas da estrutura dos agentes desenvolvidos, ajudando no posterior mapeamento do *framework* proposto.

Para a representação semântica do conhecimento, foram apresentadas as ontologias. No *framework* proposto, ontologias são usadas na representação do conhecimento

disponível no agente, para transmitir e solicitar conhecimentos, e para a descrição dos objetivos.

Por último, o estudo e conceituação de aspectos relacionados a GC deram o embasamento teórico necessário para o mapeamento de um processo de GC em uma arquitura de software. As atividades do processo de GC sintetizado, por exemplo, foram mapeadas como procedimentos na arquitetura proposta. Aspectos da técnica de RBC também foram incorporados ao modelo, como é o caso do uso de uma base de conhecimento composta por diversas entidades de conhecimento que encapsulam todo o conhecimento necessário para que seja atingido determinado objetivo - entidades que serão referenciadas no decorrer do texto como *objetos de conhecimento*.

3 ORGANIZAÇÃO DO CONHECIMENTO EM AGENTES DE SOFTWARE

3.1 Problema

Sistemas multiagentes geralmente são considerados complexos em relação a sua estrutura e funcionalidade (WEISS, 1995). Em muitas aplicações, até mesmo naquelas em que o ambiente parece extremamente simples, é difícil ou mesmo impossível, determinar corretamente, em tempo de projeto, os possíveis comportamentos e atividades realizados pelos SMAs. Para fazer este tipo de previsão, seria necessário ter conhecimento, por exemplo, de quais os requisitos do ambiente que poderiam aparecer no sistema em tempo de execução e também de como os agentes disponíveis no sistema iriam interagir para atender esses requisitos. Este tipo de problema remete-nos a questões relacionadas ao aprendizado (ou aquisição de novos conhecimentos) e, se o agente for capaz de aprender, ele deve ser capaz, também, de representar o conhecimento adquirido para que ele possa ser reusado ou até mesmo distribuído para outros agentes.

Para que os agentes possam aprender com a sua experiência, incorporando fatos e conceitos, é necessário que as plataformas para desenvolvimento de SMAs ofereçam formas de representação e estruturação do conhecimento, permitindo aos agentes efetuar transformações sobre estas estruturas. Porém, através da descrição das plataformas disponíveis para o desenvolvimento de SMAs (Seção 2.3.1) pôde-se observar a falta de suporte nativo à definição interna dos agentes, principalmente em relação à criação e organização do conhecimento. Assim, este trabalho objetiva contribuir para o avanço da tecnologia de agentes, permitindo o gerenciamento do conhecimento disponível em agentes de software, através do mapeamento dos elementos envolvidos em um processo de GC para um modelo de arquitetura de software.

O *framework* proposto, quando integrado a uma plataforma de desenvolvimento de SMAs, permite que os agentes gerenciem seus conhecimentos internos, podendo criar, adquirir, distribuir, aplicar e armazenar conhecimento. O compartilhamento ou troca de conhecimento social (conhecimento disponível em *yellow pages*) não é considerado

no *framework*, pois se considera que este tipo de conhecimento deve ser gerenciado pela própria plataforma de desenvolvimento, estando acessível a todos os agentes do sistema.

Com o objetivo de apresentar o estado da arte em gerenciamento de conhecimento de sistemas, a seguir serão apresentados alguns trabalhos relacionados ao aqui proposto.

3.2 Trabalhos relacionados

Na literatura foram encontradas várias arquiteturas de software para GC corporativa, como (HOUARI e FAR, 2004; WEISS, 1999; LAWTON, 2001), que têm como objetivo expandir o alcance e potencializar a velocidade de transferência do conhecimento. Porém, não foram encontradas arquiteturas que utilizam, de forma completa e integrada, um processo de GC com o objetivo de gerir o conhecimento interno do sistema, fato que é a motivação para a proposta deste trabalho.

Assim, a seguir serão detalhadas algumas propostas de abordagens que contemplam ao menos uma das atividades do processo de GC sintetizado (apresentado na Seção 2.4.1) ou que demonstram o uso de tecnologias como RBC e ontologias aplicadas a GC ou SMAs. Ao final do capítulo, na Seção 3.3, é mostrada uma tabela comparativa entre as abordagens descritas, identificando de que modo são tratadas as atividades do processo em cada trabalho.

3.2.1 Compartilhamento de conhecimento em SMAs

Rybinski e Ryzko (2003) apresentam um trabalho sobre o compartilhamento de conhecimento em SMAs. Neste trabalho, como será detalhado a seguir, pode-se identificar aspectos de duas das atividades do processo de GC sintetizado: a atividade capturar e a atividade distribuir.

No artigo são descritos aspectos referentes à arquitetura dos agentes e à arquitetura do SMA. Em relação à arquitetura dos agentes, cujo sistema lógico é definido pelos autores como “um sistema de raciocínio padrão baseado no conhecimento local e extensível através do conhecimento de agentes externos”, são descritos três aspectos: a arquitetura da base de dados, a classificação de fatos e regras e o processo de raciocínio.

A base de dados é composta por um conjunto de fatos sobre o mundo, por regras que representam o conhecimento do domínio e por restrições semânticas. Já o conhecimento interno do agente é dividido em dois tipos de conhecimento: de domínio e de ambiente. O conhecimento de domínio contém todos os fatos e regras sobre o domínio no qual o agente opera. O conhecimento de ambiente, por sua vez, contém informações do ambiente em que o agente está trabalhando (é nesse tipo de conhecimento, por exemplo, que estão armazenadas informações sobre quais os outros agentes que atuam no ambiente).

Para o processo de raciocínio, todos os agentes possuem um mecanismo que permite inferir novos fatos e também invalidar sentenças lógicas consideradas falsas com uma lógica pré-definida. Como é possível que um agente não contenha conhecimento suficiente para raciocinar acerca dele mesmo, há um mecanismo global de solicitações a partir do qual são aplicadas questões aos outros membros da comunidade.

Em relação à arquitetura do SMA, são descritos dois aspectos: interação e raciocínio. Quanto à interação, que é feita em diferentes níveis, um agente só compartilha conhecimento depois de receber informações sobre a organização do sistema e sobre os estados e objetivos dos outros agentes. Também, apenas são compartilhadas informações com os agentes mais confiáveis (cada agente classifica os outros membros da comunidade de acordo com níveis de confiança - confiável, honesto e outros).

Quando um agente necessita de algum conhecimento, ele deve executar uma seqüência de passos. Primeiro, deve ser verificado o conhecimento disponível na base de dados local. Conhecimento classificado como honesto ou outro só deve ser usado caso não seja encontrado nenhum conhecimento nativo ou confiável que atenda a solicitação (o conhecimento também é classificado em níveis, que são: nativo, confiável, honesto e outros). Se não for encontrado nenhum conhecimento compatível na base de dados, o agente deve requisitar conhecimento para os outros agentes do sistema. Agentes considerados confiáveis devem ser notificados primeiro, seguidos pelos honestos e outros.

No artigo é proposta ainda uma abordagem para diminuir o número de interações entre os agentes. Essa abordagem consiste na verificação das mensagens enviadas pelos membros da comunidade a fim de identificar quais deles estão aptos a responder uma determinada solicitação. A solicitação é enviada somente para os membros considerados aptos.

3.2.1.1 Considerações

Embora no trabalho descrito sejam considerados vários aspectos referentes a captura e distribuição de conhecimento em SMAs, alguns pontos ficam nebulosos. Um desses pontos diz respeito ao mecanismo para seleção de conhecimento. Questões do tipo "Como é escolhida a solução mais apropriada (melhor solução)?" e "Quando um conhecimento é considerado compatível com uma necessidade?" não são tratadas no artigo.

Também, como as solicitações são executadas localmente (o agente que recebe a solicitação responde com uma ou mais possíveis soluções), não há referência à transferência de conhecimento entre os agentes, o que permitiria que a solução fosse adquirida pelo próprio agente solicitante.

Em relação às contribuições dadas pelo trabalho descrito ao aqui proposto, destaca-se o processo de captura de conhecimento. De maneira similar à descrita no artigo, no *framework* proposto, através da implementação de alguns dos pontos de flexibilidade, podem ser verificados aspectos referentes à confiança na seleção de um objeto de conhecimento para aplicação e também no envio de mensagens de solicitação de conhecimento.

3.2.2 Gestão de conhecimento em sistemas computacionais inteligentes

Weber e Wu, em (WEBER e WU, 2004), descrevem uma arquitetura para projetar sistemas computacionais inteligentes (*Computational Intelligence Systems* - CIS) que

inclui um *framework* para GC cujas metodologias base são raciocínio baseado em casos e distribuição monitorada (*monitored distribution*). De acordo com os autores, este *framework* permite que o sistema aprenda a partir de suas próprias experiências podendo assim evoluir. Para avaliar o seu próprio desempenho, o sistema utiliza métricas de avaliação de execução.

A arquitetura do *framework* para GC descrita no artigo é composta por dois módulos principais, sendo que ambos os módulos consistem de um recipiente de bases de casos e de quatro processos de GC, denominados *creator*, *distributor*, *reuser* e *understander*. A Figura 3.1 apresenta a arquitetura proposta. O módulo superior é projetado para descrever experiências de execução do CIS (execuções terminadas) na base de casos principal (MCB – *Main Case Base*). O módulo inferior ou módulo das lições aprendidas (*lessons-learned*) descreve lições aplicáveis no sistema (experiências úteis para tarefas individuais) na base de lições aprendidas (LLB – *Lessons-learned Base*).

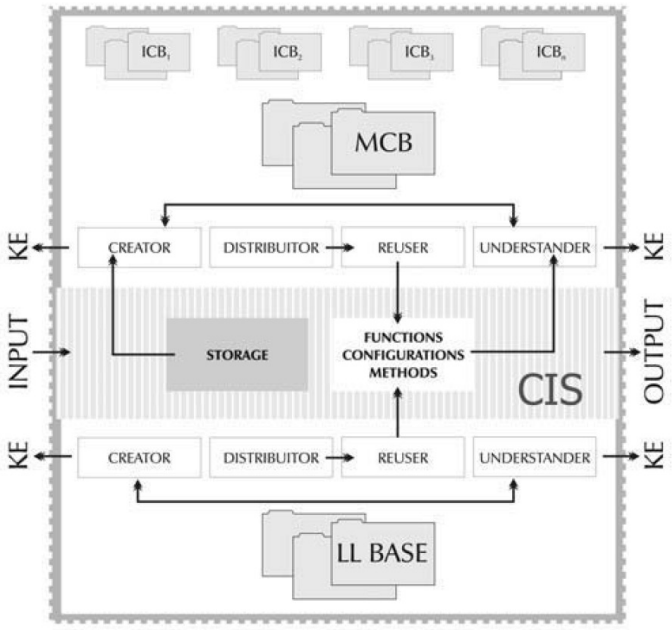


Figura 3.1: CIS + *Framework* de Gestão de Conhecimento (WEBER e WU, 2004)

O módulo superior possui ainda, como pode ser visualizado na Figura 3.1, bases de casos individuais (ICB – *Individual Case Bases*). Essas bases são adicionadas se, para uma mesma saída, o CIS oferece várias opções de métodos (cada ICB armazena uma variante de cada método). Também, há os engenheiros de conhecimento (KE – *Knowledge Engineers*) que verificam (validam) o conhecimento criado e organizado pelo sistema.

Os quatro processos de GC que compõem os módulos principais do *framework* para GC são descritos da seguinte forma:

- *Creator*: é o processo responsável pela aquisição de dados para popular as bases de casos. Nas MCBs e ICBs são coletados os dados referentes a entradas, parâmetros usados e produzidos durante a execução e saídas do CIS.
- *Understander*: refere-se a todos os métodos desenvolvidos para preparar os dados adquiridos pelo processo *Creator*.
- *Distributor*: é o processo responsável por recuperar conhecimento relevante e aplicável a todas as necessidades de conhecimento do CIS. A recuperação de conhecimento é feita através de parâmetros de similaridade definidos no processo *Understander*.
- *Reuser*: este processo dispara comandos contidos no conhecimento recuperado pelo processo *Distributor*. Também, é neste processo que são acionadas as necessidades de adaptação do conhecimento recuperado.

A integração do *framework* em um CIS é feita de acordo com o ciclo de vida do *framework*. São três os estágios do ciclo: infância, adolescência e maturidade. Nos primeiros dois estágios, há um controle total dos engenheiros de conhecimento. Na infância, métodos e processos são testados e as primeiras experiências são armazenadas; na adolescência o sistema começa a tomar decisões; por último, no estágio maduro, há pouca interferência humana e, se o sistema necessitar de alguma manutenção, métodos próprios de manutenção irão determinar e sinalizar a sua necessidade.

De acordo com os autores, acredita-se que este *framework* possa ser utilizável em outros tipos de sistemas computacionais, mas não há qualquer experimento que comprove sua aplicação em outros ambientes.

3.2.2.1 Considerações

Em Weber e Wu (2004) são encontrados aspectos de grande parte das atividades do processo de GC sintetizado. O processo *Creator*, por exemplo, está relacionado

com a atividade capturar/criar do processo de GC guia. Já o processo *Understander* poderia ser mapeado na atividade organizar. Porém, embora sejam citadas várias questões envolvendo as atividades do processo, muitas delas estão descritas de forma superficial, o que dificulta o entendimento de que tipo de tecnologia ou método é de fato utilizado. Por exemplo, após o conhecimento ser adquirido, sabe-se que ele passa por uma preparação no processo *Understander*, mas não se sabe qual o tipo de representação de conhecimento utilizado. A definição das métricas de manutenção e dos parâmetros de similaridade (que indicam a similaridade entre um caso candidato e o caso de entrada) também não é apresentada.

O trabalho descrito, mesmo não tratando especificamente de GC em agentes de software, traz algumas contribuições para o trabalho que está sendo aqui proposto. Como contribuição conceitual, destaca-se a idéia de evolução a partir do conhecimento adquirido (com o uso de métricas de avaliação de desempenho, os agentes podem analisar suas execuções e sinalizar a necessidade de um conhecimento mais adequado, que tende a atender de forma mais satisfatória os seus objetivos).

Entre as principais diferenças entre o trabalho aqui proposto e o apresentado por Weber e Wu, está o tipo de conhecimento ou o conteúdo dos casos armazenados. No trabalho descrito são armazenados, principalmente, aspectos de configuração do CIS (entradas, parâmetros usados e produzidos) e as saídas geradas. No *framework* aqui proposto, a idéia é armazenar nos objetos de conhecimento todo o conhecimento necessário para atingir determinado objetivo, o que inclui, por exemplo, o código das ações que deverão ser executadas para que o objetivo possa ser satisfeito.

3.2.3 Uso de ontologias para o compartilhamento de conhecimento

Pesquisadores da área de Inteligência Artificial estão adotando ontologias como um formalismo de representação de conhecimento compreensível para prover raciocínio de senso comum no suporte a tarefas como aquisição e reuso de conhecimento (WEBER e KAPLAN, 2003). Na literatura há vários trabalhos que utilizam ontologias para o fim citado, como (FERNANDES *et al.*, 2003; DAVIES *et al*., 2002).

Fernandes e co-autores, em (FERNANDES *et al*., 2003), apresentam uma abordagem baseada em ontologias para organizar, compartilhar e consultar objetos de conhecimento na *Web*. O sistema proposto fornece um modelo de dados que auxilia os usuários a descrever seus objetos de conhecimento pessoais (como exemplo de objetos de conhecimento tem-se "oql.pdf" e "www.google.com", que correspondem, respectivamente, a um arquivo e a um endereço *Web*), que podem ser mais tarde compartilhados com outros usuários. A busca por objetos de conhecimento é feita a partir de consultas nos meta-dados que descrevem o objeto, como será descrito a seguir.

Para integrar os objetos de conhecimento dos usuários, o sistema foi projetado de acordo com duas perspectivas: a arquitetura física, que especifica os componentes do sistema e a sua integração; e a arquitetura lógica, que organiza os objetos do sistema em três visões lógicas.

Os componentes do sistema são distribuídos dentro de dois módulos, o local e o do servidor. Conforme indicado na Figura 3.2, no módulo local há uma ferramenta para a criação de uma taxonomia de domínio que é usada para descrever e classificar os objetos de conhecimento. A definição de uma taxonomia local dá aos usuários uma forma personalizada de descrever seus objetos. Já o suporte a definição de meta-dados possibilita a descrição, a organização e o relato de documentos e *links*, promovendo interoperabilidade e auxiliando nos processos de consulta e recuperação de conhecimento.

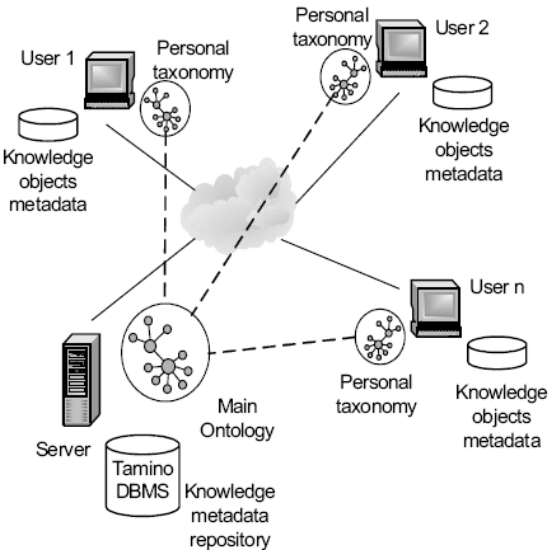


Figura 3.2: Arquitetura física do sistema (FERNANDES *et al*., 2003)

Uma vez que os objetos de conhecimento do usuário estão classificados e descritos utilizando a taxonomia local, eles podem ser publicados no módulo servidor, que pode ser visto como um repositório de meta-dados. Assim, os objetos ficam aptos para serem compartilhados entre os usuários. O módulo servidor hospeda, além dos meta-dados dos objetos, uma ontologia global que integra diferentes taxonomias através de um vocabulário comum.

A arquitetura lógica é estruturada de acordo com três camadas: interna, de dados e semântica. A distribuição em camadas pode ser vista na Figura 3.3. A camada interna corresponde à representação física dos objetos de conhecimento. A camada de dados está relacionada aos meta-dados dos objetos de conhecimento, expressados em RDF. Por último, a camada semântica permite ao usuário criar uma ontologia personalizada para classificar os objetos de uma maneira mais flexível. Segundo os autores, com o uso de ontologias, diferentemente de sistemas com diretórios de arquivos tradicionais, os objetos podem ser classificados em diferentes categorias, o que torna mais fácil o processo de organização. Também, é possível estabelecer relacionamentos entre diferentes categorias, o que pode ser explorado por mecanismos de busca.

Segundo os autores, a interoperabilidade do sistema é suportada pela integração de várias ontologias, onde as classes das ontologias pessoais dos usuários são mapeadas na ontologia global, que funciona como um vocabulário compartilhado. O processo de mapeamento é *um-a-um*, ou seja, para cada conceito da ontologia local deve ser identificado um conceito correspondente na ontologia global. Com este mapeamento, quando um usuário submete uma consulta usando seu vocabulário personalizado, o sistema irá retornar não apenas os objetos classificados na sua categoria local, mas todos os objetos relacionados ao conceito identificado na ontologia global. Por exemplo, quando um usuário faz uma consulta com o vocabulário "DB" são retornados todos os objetos relacionados com o conceito "Database" da ontologia global ("DB" está mapeado como "Database").

Em Davies *et al.* (2002) é descrito o OntoShare. OntoShare é um ambiente *Web* de compartilhamento de conhecimento baseado em ontologias para comunidades de prática[5]. No OntoShare, os interesses dos usuários são armazenados em *perfis de usuários*.

[5] Comunidades de prática são grupos dentro (ou através de) organizações que compartilham um conjunto comum de informações.

Um perfil de usuário contém um conjunto de tópicos ou conceitos ontológicos (classes declaradas em RDFS) onde o usuário declara seus interesses.

Na medida em que os usuários contribuem com informações para a comunidade, são criados novos recursos de conhecimento anotados com meta-dados. Recursos de conhecimento podem conter, por exemplo, uma nota (comentário) do usuário, uma página *Web* ou informações copiadas de outras aplicações ou do próprio computador do usuário.

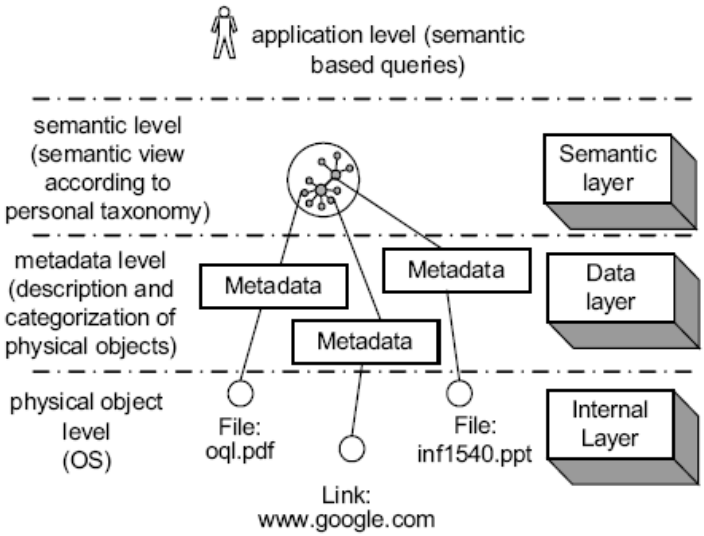


Figura 3.3: Arquitetura lógica do sistema (FERNANDES *et al*., 2003)

O OntoShare é usado para armazenar, recuperar, sumarizar e notificar usuários sobre determinadas informações compartilhadas por algum usuário da comunidade. O compartilhamento é feito através de um cliente Java, a partir do qual o usuário pode inserir meta-dados para a nova informação disponibilizada. A princípio, o próprio sistema sugere um conjunto de conceitos que podem estar relacionados ao conteúdo submetido. O usuário pode então aceitar as recomendações do sistema ou modificá-las sugerindo alternativas ou conceitos adicionais que estão associados ao conteúdo.

Depois da inserção dos meta-dados, o OntoShare cria um resumo do novo conteúdo para comparar com os perfis de interesse dos usuários da comunidade. Toda vez que é identificada uma alta similaridade entre os termos do novo conteúdo (ou documento submetido) e algum perfil de usuário, o OntoShare gera uma notificação indicando a localização do novo conteúdo, quem o está compartilhando e quais foram os meta-dados adicionados.

No perfil do usuário que compartilhou o novo conteúdo, também é verificada a existência dos conceitos relevantes inseridos nos meta-dados do conteúdo. Caso seja encontrado nos meta-dados algum conceito não cadastrado nos interesses do usuário, o OntoShare sugere ao usuário que este conceito seja adicionado à seu perfil. Esta característica é apresentada como uma capacidade de aprendizado do OntoShare.

Para a descrição dos conteúdos é utilizada uma outra ontologia. Nesta ontologia há informações sobre palavras-chave associadas, o resumo do documento, seu título, a anotação fornecida pelo usuário, sua URL, o responsável por sua submissão e a data de armazenamento.

3.2.3.1 Considerações

A descrição dos trabalhos de Fernandes *et al.* (2003) e Davies *et al.* (2002) permite a identificação de possíveis usos de ontologias para auxiliar a Gestão de Conhecimento. Ontologias possibilitam, como pôde ser visto, a criação de um vocabulário compartilhado, o uso de buscas baseadas em semântica, a atribuição de meta-dados aos recursos de conhecimento, entre outros.

Muito embora os trabalhos citados não estejam relacionados ao gerenciamento de conhecimento em sistemas computacionais, vários aspectos relacionados ao uso de ontologias para descrever e consultar os objetos de conhecimento serão utilizados no trabalho aqui proposto. Em relação à busca baseada em semântica, por exemplo, da mesma forma que o descrito nos trabalhos, a busca por objetos de conhecimento para atender a uma determinada consulta será feita sobre informações presentes na estrutura ontológica que descreve o objeto. No presente trabalho, cada objeto de conhecimento está associado a uma ontologia - que descreve os conceitos do domínio ao qual se aplica o conhecimento contido no objeto - e as consultas (ou requisições de conhecimento) são também expressas em estruturas ontológicas. Salienta-se, no entanto, que os aspectos utilizados dos trabalhos citados não estão relacionados, de maneira alguma, ao conteúdo dos objetos de conhecimento. Nos trabalhos descritos, os objetos de conhecimento têm referência a um documento ou *link*, já no aqui proposto, eles são uma agregação de vários itens que permitem que um agente alcance determinado objetivo.

Assim, entre as principais diferenças entre os trabalhos descritos e o aqui proposto, destacam-se:

- Conteúdo dos objetos de conhecimento: nos trabalhos descritos, como há o compartilhamento de conhecimento entre indivíduos, os objetos de conhecimento são constituídos de documentos ou *links*
- Existência de uma taxonomia local: em Fernandes *et al*. (2003) cada usuário está relacionado a uma taxonomia local a partir da qual são retirados os meta-dados que descrevem todos os seus objetos de conhecimento. No trabalho aqui proposto, cada objeto de conhecimento está associado a sua própria ontologia e não há nenhuma relação explícita entre os diferentes objetos de conhecimento de um mesmo agente.

3.2.4 Raciocínio baseado em casos em SMAs

Na literatura foram encontrados alguns estudos que utilizam a técnica de raciocínio baseado em casos em SMAs, principalmente com o objetivo de possibilitar o aprendizado (ou aquisição de novos conhecimentos) de agentes de software (Plaza e Ontanón, 2003; Plaza, 2005). Devido à alta similaridade e o fácil mapeamento entre os passos/etapas da técnica de raciocínio baseado em casos e os processos de GC, a seguir serão descritos, brevemente, alguns dos estudos identificados.

Em Plaza e Ontanón (2003) é apresentada uma abordagem para SMAs onde os agentes são capazes de resolver problemas e aprender através de RBC. Neste estudo, cada agente possui sua própria base de casos e há protocolos de interação para possibilitar que os agentes colaborem entre si, compartilhando diferentes casos. Já em Plaza (2005), é apresenta a técnica CoopCA, que é uma técnica de adaptação construtiva em SMAs para reuso de casos. Desta vez, a construção dos casos é feita de forma colaborativa entre os agentes do sistema (o processo de adaptação é considerado, pelo autor, aberto e distribuído), pois todos os agentes podem sugerir possíveis componentes para adicionar ao caso.

Também, há estudos que utilizam a técnica de RBC para formalizar o conhecimento adquirido com o monitoramento das ações do usuário, gerando assim novos

casos para popular a base de casos. Em Al-Sakran (2006), por exemplo, é proposta uma arquitetura que permite aos usuários coletar, compartilhar, distribuir e reusar objetos de aprendizado vindos de bases de conhecimento heterogêneas. Neste estudo, a técnica de RBC faz uso de experiências passadas de outros estudantes, recuperadas a partir da base de casos, para resolver as novas questões que surgem.

3.2.4.1 Considerações

Independente da meta, a tarefa que os sistemas de RBC executam é a comparação entre um novo caso alvo e os casos armazenados na base de casos. Esta é a essência da interpretação: identificar o que é relevante ao avaliar a similaridade e ordenar os resultados. É essa característica que torna as aplicações baseadas em RBC apropriadas para a automatização de processos de GC em sistemas inteligentes.

Como pode ser visto na descrição dos trabalhos na subseção anterior, a técnica de RBC é utilizada em agentes de software principalmente no sentido de permitir a adaptação e a aprendizagem dessas entidades. Isso é justificado, pois, de acordo com Aamodt e Plaza (1994), é mais fácil aprender a partir da experiência na resolução de um problema concreto do que generalizar a partir dela.

De qualquer forma, independentemente da necessidade ou não de adaptar os casos para o uso em uma determinada situação, o objetivo geral da técnica de RBC é o suporte aos processos de armazenamento e recuperação de conhecimento. Em se tratando de armazenamento de conhecimento, têm-se o problema do tipo de representação dos casos. Este problema refere-se, fundamentalmente, a que tipos de informações devem ser armazenadas em um caso, qual a estrutura apropriada para descrever o conteúdo de um caso e como organizar e indexar a memória de casos para uma efetiva recuperação e reutilização.

Dentre os trabalhos descritos, apenas um faz referência ao tipo de representação de casos utilizado: em (AL-SAKRAN, 2006) os casos são representados através de uma lista de atributos valorados, que um dos tipos de representação mais comum utilizado para representar casos em RBC.

No trabalho aqui proposto, vários aspectos da técnica de RBC são considerados. Dentre esses aspectos destaca-se a existência de uma base de conhecimento composta por objetos de conhecimento (que poderiam ser considerados casos), a avaliação da similaridade entre os objetos para a utilização em uma determinada necessidade, a representação dos objetos de acordo com um tipo definido de representação e o armazenamento de novos casos como resultado de uma lição aprendida (processo similar ao descrito em (AL-SAKRAN, 2006) - a execução dos agentes é monitorada para a aquisição de novos conhecimentos).

3.3 Considerações sobre o capítulo

Como apresentado nas seções anteriores, diversas são as abordagens propostas para dar apoio aos processos de GC, cada qual com suas contribuições e limitações. Desta forma, se torna conveniente a combinação de diferentes abordagens, como, por exemplo, a técnica de raciocínio baseado em casos e as ontologias. A Tabela 3.1 apresenta um comparativo com as principais características das abordagens descritas. Na tabela são destacados os seguintes aspectos:

- Suporte às atividades de processo de GC utilizado como guia: verificação de quais atividades são cobertas, de forma parcial ou integral, pela abordagem proposta.
- Tipo de representação de conhecimento: o conhecimento capturado deve ser convertido para uma linguagem de representação de conhecimento, para mais tarde ser recuperado e distribuído. Assim, neste item é verificado qual o tipo de representação de conhecimento sugerido pela abordagem.
- Conteúdo dos recursos de conhecimento: verificação de quais tipos de conteúdos que são indexados pelos recursos de conhecimento sugeridos pela abordagem.
- Algoritmo para verificação da similaridade: indica como o conhecimento

disponível no sistema é recuperado quando é recebida uma solicitação ou consulta.

Admitindo-se a natureza subjetiva de alguns dos itens identificados, como a cobertura das atividades do processo de GC, procurou-se verificar, nas abordagens apresentadas, indicativos de tecnologias ou mecanismos que servissem como meio para prover os aspectos requeridos, ou seja, a existência de características que indiquem a preocupação com tais fatores. Na Tabela 3.1, não há referência ao quarto item descrito (algoritmo para verificação de similaridade), porque, embora a maioria das abordagens cite a verificação da similaridade no momento da distribuição do conhecimento, não há qualquer indicativo de que tipo de algoritmo é utilizado para essa verificação ao longo da descrição dos trabalhos.

Como pode ser observado na Tabela 3.1, nenhuma das abordagens provê suporte a todos os aspectos considerados. A abordagem mais abrangente, em se tratando das atividades do processo de GC, é a que trata de GC em sistemas computacionais inteligentes (WEBER e WU, 2004). No entanto, essa abordagem não é aplicada ou ao menos contextualizada em agentes de software. Neste contexto, identifica-se uma carência de pesquisas que trabalhem com GC em agentes de software, considerando as características próprias dessas entidades.

Tabela 3.1: Comparação das abordagens descritas

		Trabalhos Relacionados				
		Compartilhamento de Conhecimento em SMAs	Gestão de conhecimento em sistemas computacionais inteligentes	Uso de ontologias para o compartilhamento de conhecimento		Raciocínio baseado em Casos em SMAs
				Fernandes *e co-autores*	Davies *e co-autores*	
Atividades do Processo de GC	Capturar/Criar	Os agentes solicitam conhecimento aos outros membros da comunidade.	Através do processo *Creator* são adquiridos dados para criar novos casos.	São feitas consultas nos meta-dados que descrevem os objetos de conhecimento.	Não indica.	Há compartilhamento de casos entre diferentes agentes.
	Organizar	Não indica.	No processo *Understander* é organizado o conhecimento criado pelo sistema.	Os recursos de conhecimento (ou objetos de conhecimento) são anotados com meta-dados.		O conhecimento é estruturado em casos.
	Armazenar	Cita a existência de uma base de dados.	Há três tipos de bases de casos diferentes.	Há uma base de meta-dados de objetos de conhecimento.		Há bases de casos.
	Distribuir	A distribuição é feita através de uma comparação entre o conhecimento local e a solicitação.	O conhecimento é recuperado através do processo *Distributor*.	Os objetos de conhecimento podem ser compartilhados entre os usuários.	O sistema notifica os usuários sobre informações compartilhadas.	Há compartilhamento de casos entre diferentes agentes.
	Aplicar	Não indica.	Através do processo *Reuser* são disparados os comandos contidos no conhecimento recuperado.	Não se aplica.		Não indica.
Tipo de representação de conhecimento		Não indica.	Não indica.	Os meta-dados são representados em ontologias (RDF e RDFS)		O conhecimento é representado em casos (a estrutura dos casos não é explícita).
Conteúdo dos recursos de conhecimento		Acredita-se que é composto por fatos, regras e restrições (não está explícito).	Aspectos de configuração do CIS (entradas, parâmetros usados e produzidos) e saídas geradas.	Documentos e *links*		Não indica.

4 MODELO DE ARQUITETURA PARA GESTÃO DE CONHECIMENTO EM AGENTES DE SOFTWARE

Tendo em vista a necessidade de organização do conhecimento de agentes de software e a carência de pesquisas na área, neste trabalho está sendo proposto um *framework* cujo principal objetivo é permitir aos agentes formalizar o conhecimento disponível e também transmitir esse conhecimento para que ele possa ser reusado. A arquitetura proposta conta com procedimentos para o mapeamento de cada uma das atividades do processo de GC sintetizado, ou seja, procedimentos para criar, organizar, armazenar, distribuir, capturar e aplicar os objetos de conhecimento.

No *framework*, um objeto de conhecimento é uma entidade que possui todos os elementos requeridos para resolver determinado problema, definição similar a "*knowledge source*" dada em (FERBER, 1999). Todas as regras e fatos de um agente podem estar relacionados a um ou mais objetos de conhecimento. Os agentes podem usar esses objetos para realizar suas tarefas ou para gerar novos conhecimentos. Neste trabalho, o termo conhecimento, da mesma forma que em Kucza (2001), refere-se à "informação em ação", isto é, informação aplicada para um propósito.

A apresentação do *framework* será feita em três diferentes níveis. No primeiro nível, serão apresentados aspectos referentes às atividades do processo de GC sintetizado. A análise das atividades, além de permitir a verificação da uniformidade do fluxo apresentado inicialmente e a identificação de outros aspectos de um processo (como entradas e saídas), visa esclarecer os requisitos necessários para a aplicação de um processo de GC no contexto de uma arquitetura de software.

No segundo nível, é descrito o modelo conceitual do *framework* e o seu mapeamento teórico sobre duas plataformas, o JADE e o SemantiCore. Já no terceiro nível (que será apresentado no Capítulo 5), é fornecida uma descrição completa, em termos de implementação, do *framework* proposto já integrado ao SemantiCore.

Imaginando-se o *framework* proposto como uma camada ou componente a ser integrado no agente, muitas vezes, visando acomodar os diferentes estilos arquiteturais de um agente, serão encontradas no texto referências às outras partes ou componentes do agente.

4.1 Análise do processo de Gestão de Conhecimento

Durante a verificação da aplicabilidade do processo de GC sintetizado em uma arquitetura de software, verificou-se a necessidade de duas mudanças no processo. A primeira mudança refere-se à separação da atividade capturar/criar em duas atividades, onde:

- Atividade capturar: refere-se ao procedimento de busca por conhecimento para uma determinada necessidade. Essa busca pode ser interna (na base de conhecimento do próprio agente) ou externa (na base de conhecimento global ou por solicitação aos outros agentes do sistema).
- Atividade criar: refere-se à solicitação de encapsulamento ou organização do conhecimento novo disponível no agente.

A segunda mudança diz respeito à criação de três fluxos distintos de atividades. Essa divisão é justificada pela verificação de que, por exemplo, nem todo o conhecimento armazenado deve ser posteriormente distribuído, pois a solicitação de conhecimento não está associada, em primeira instância, ao armazenamento do conhecimento, mas sim a uma necessidade de conhecimento recebida.

A Figura 4.1 mostra as atividades dispostas nos três fluxos, onde: o Fluxo 1 é o responsável por possibilitar a criação de novos conhecimentos (estruturados em objetos de conhecimento); o Fluxo 2 permite que haja compartilhamento de conhecimento entre diferentes agentes; e o Fluxo 3 sinaliza uma necessidade de conhecimento do agente, aplicando e armazenando o conhecimento mais similar recebido (a similaridade é verificada em relação a necessidade de conhecimento sinalizada). Salienta-se que, tanto o Fluxo 1 quanto o Fluxo 3 terminam com a execução da atividade armazenar, pois ambos envolvem a aquisição de novos conhecimentos, seja pela experiência ou pelo recebimento de conhecimento externo ao agente. Também, no Fluxo 3 não é necessária a organização do conhecimento recebido, visto que o conhecimento compartilhado já está estruturado como um objeto de conhecimento.

A seguir serão apresentados aspectos relacionados a cada uma das atividades do processo. A idéia é identificar as pré-condições (ou eventos) que desencadeiam a execução de cada atividade, as pós-condições, que são os resultados atingidos com o término da

atividade, e também os tipos de dependência para a execução da atividade. Há dois tipos de dependência: interna e externa. Se a dependência for interna, significa que a atividade é disparada pelo recebimento de algum tipo de sinalização ou solicitação do agente onde o *framework* está integrado. Já a dependência externa é caracterizada pelo recebimento de mensagens pelo ambiente, ou seja, por solicitações de outros agentes do sistema.

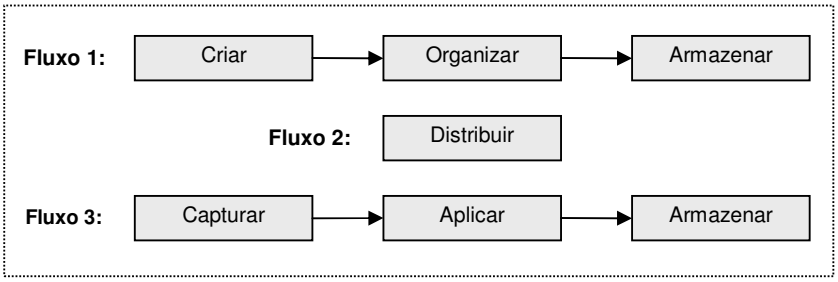


Figura 4.1: Fluxos de atividades do processo de GC

4.1.1 Fluxo 1 – Criação e representação de novos conhecimentos

Toda vez que é recebida uma indicação de que há conhecimento novo disponível no agente é iniciada a atividade **criar**. É através dessa atividade que é verificada a disponibilidade de informações para que haja, de fato, a criação de um novo objeto de conhecimento. Essa atividade tem como características:

- Entrada: indicação de conhecimento novo disponível.
- Saída: dados para a criação do objeto de conhecimento disponíveis.
- Dependência: interna, visto que a indicação de conhecimento novo disponível vem de algum componente do agente.

A atividade **organizar** é responsável por encapsular o conhecimento disponível em um objeto de conhecimento. Também, é nesta atividade que são definidas as restrições de acesso ao objeto. Essa atividade tem como características:

- Entrada: dados para criação do objeto de conhecimento.
- Saída: objeto de conhecimento.

- Dependência: não há (o disparo desta atividade é feito ao final da atividade criar).

Na atividade **armazenar,** o objeto de conhecimento deve ser registrado e armazenado na base local de conhecimento. Essa atividade tem como características:

- Entrada: objeto de conhecimento.
- Saída: objeto de conhecimento armazenado.
- Dependência: não há (o disparo desta atividade é feito ao final das atividades organizar e aplicar).

A Figura 4.2 mostra, através de uma representação esquemática, o que acontece cada vez que é iniciado o Fluxo 1. Como pode ser observado na figura, a execução inicia com a chegada de um mensagem indicando a detecção de um novo conhecimento no agente. Essa mensagem é processada pela atividade criar, que então verifica as informações necessárias para a criação do novo conhecimento. De posse das informações recuperadas, é criado um novo objeto de conhecimento (KO – *Knowledge Object*) na atividade organizar. Por último, o novo objeto é armazenado na base de conhecimento (através da atividade armazenar).

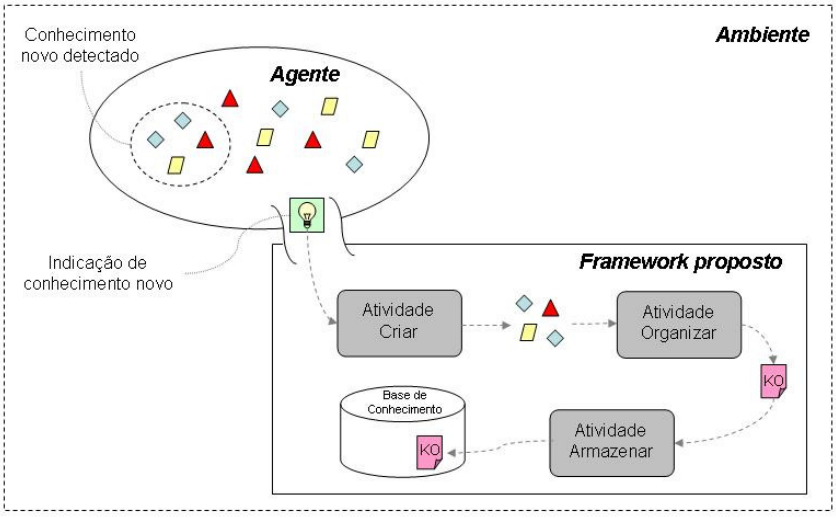


Figura 4.2: Modelo esquemático da execução do Fluxo 1

4.1.2 Fluxo 2 – Compartilhamento de conhecimento

Quando um agente envia uma solicitação de conhecimento, todos os agentes que recebem a solicitação verificam suas bases locais de conhecimento em busca de objetos de conhecimento para atender a solicitação. Se existe algum objeto compatível, depois de verificadas as restrições de compartilhamento, é feito o envio do objeto ao agente solicitante. Esse é o papel da atividade **distribuir**. Essa atividade tem como características:

- Entrada: solicitação de conhecimento.
- Saída: objeto(s) de conhecimento enviado(s) ou não há objeto compatível com a solicitação.
- Dependência: externa, visto que a solicitação de conhecimento vem de outro agente.

Na Figura 4.3 são apresentados, de forma esquemática, alguns aspectos da execução do Fluxo 2. Conforme indicado na figura, a atividade distribuir inicia com a chegada de uma solicitação de conhecimento. Através das informações contidas nessa solicitação (estruturadas em uma ontologia) são verificados os objetos mais similares disponíveis. Se forem encontrados objetos compatíveis com a solicitação, como é o caso do cenário ilustrado, é enviada uma mensagem de "entrega" de conhecimento ao agente solicitante.

Caso não sejam encontrados objetos compatíveis com uma solicitação, não é enviada qualquer mensagem. A não compatibilidade é registrada quando nenhum dos objetos disponíveis tem um grau de similaridade maior ou igual ao limiar mínimo estabelecido. O agente pode, também, optar por não compartilhar conhecimento com determinados agentes ou com todos os agentes do sistema. Neste caso, mesmo tendo conhecimento compatível disponível, não serão enviadas mensagens de "entrega" de conhecimento.

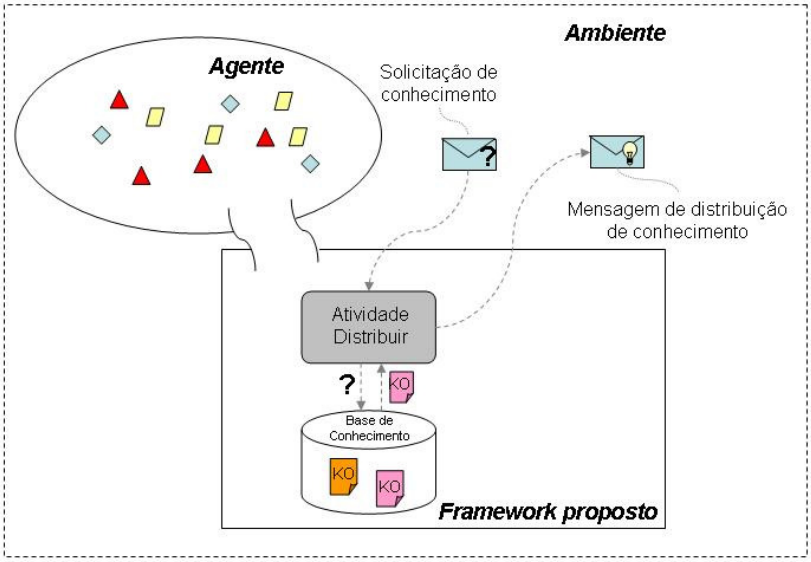


Figura 4.3: Modelo esquemático da execução do Fluxo 2

4.1.3 Fluxo 3 – Aquisição e aplicação de novos conhecimentos

Quando é percebida alguma necessidade de conhecimento no agente, o Fluxo 3 deve ser executado para que seja localizado e aplicado o conhecimento necessário. Através da atividade **capturar**, as informações referentes à necessidade de conhecimento são encapsuladas e enviadas pelo ambiente. É ainda nessa atividade que o conhecimento recebido é selecionado. Essa atividade tem como características:

- Entrada: indicação de necessidade de conhecimento.
- Saída: objeto de conhecimento compatível encontrado, ou não existe objeto de conhecimento compatível com a solicitação.
- Dependência: interna, já que a indicação de conhecimento faltante vem de outra parte do agente.

A atividade **aplicar** é responsável por ativar (ou "carregar") os itens do objeto de conhecimento selecionado na atividade capturar. Para a ativação dos itens de um objeto de conhecimento, é necessária a verificação da compatibilidade com o conhecimento já ativo no agente. Essa atividade tem como características:

- Entrada: objeto de conhecimento.
- Saída: objeto compatível e carregado, ou objeto incompatível.
- Dependência: não há (esta atividade é disparada pela atividade capturar).

Para que um agente de software dê início a um processo de aquisição de novos conhecimentos ele deve ser capaz de identificar, primeiramente, o conhecimento faltante ou ineficiente. Conhecimento ineficiente, neste caso, refere-se àquele conhecimento que, ao ser utilizado para atingir determinado objetivo, possibilita que o objetivo seja atingido parcialmente ou de maneira pouco satisfatória. Para a verificação de conhecimento ineficiente, o *framework* proposto guarda registros de execução e, a partir de uma análise desses registros, pode ser também disparada a execução do Fluxo 3 (esse mecanismo será melhor explicado na próxima seção).

A Figura 4.4 traz um esquema ilustrativo do que acontece cada vez que é iniciado o Fluxo 3, disparado através da identificação de um conhecimento faltante no agente. Como pode ser observado na figura, informações sobre a necessidade de conhecimento são encaminhadas à atividade capturar, que então as encapsula em uma mensagem para envio aos outros agentes do sistema. Depois de certo tempo, são verificadas as respostas recebidas para a solicitação, a fim de identificar o conhecimento recebido mais apropriado. O conhecimento selecionado (estruturado como um objeto de conhecimento) é encaminhado, então, para a atividade aplicar. Se todos os itens do objeto forem carregados com sucesso, o novo objeto de conhecimento é armazenado na base de conhecimento local através da atividade armazenar (descrita na Seção 4.1.1).

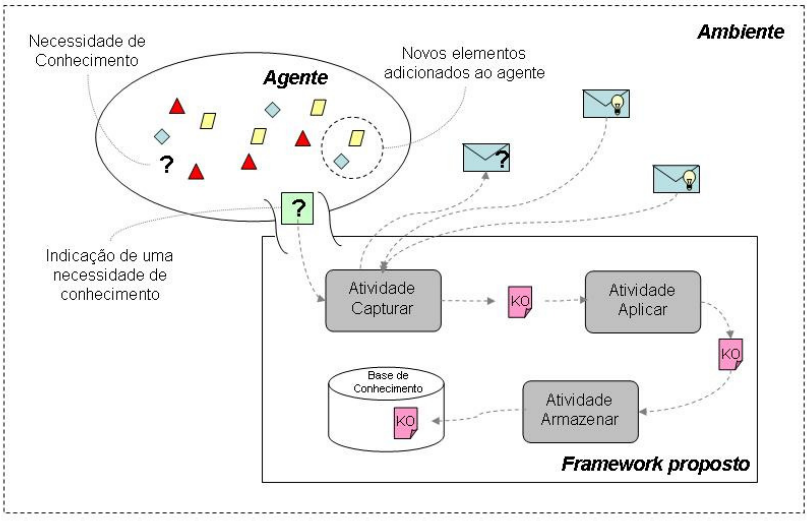


Figura 4.4: Modelo esquemático da execução do Fluxo 3

4.2 Descrição do Modelo Conceitual

A Figura 4.5 apresenta a modelo conceitual da arquitetura do *framework*. Observando-se a figura, podem-se destacar dois aspectos quanto ao domínio de aplicação no qual o *framework* proposto pode ser utilizado. Primeiro, o *framework* objetiva tratar aplicações que trabalham com conceitos de *Web* Semântica e ontologias. No modelo conceitual, isso pode ser observado, por exemplo, no atributo *ontology* da classe *KOGoal* (o tipo do atributo é *object* para permitir o uso de diferentes APIs para trabalhar com ontologias). Também, tanto o formato para representação do conteúdo de mensagens quanto o formato para a representação do próprio objeto de conhecimento remetem ao uso de ontologias, o que pode ser observado no modelo, respectivamente, no atributo *contentSchema* da classe *KnowledgeOrganizer* e no atributo *schema* da classe *KnowledgeObject*.

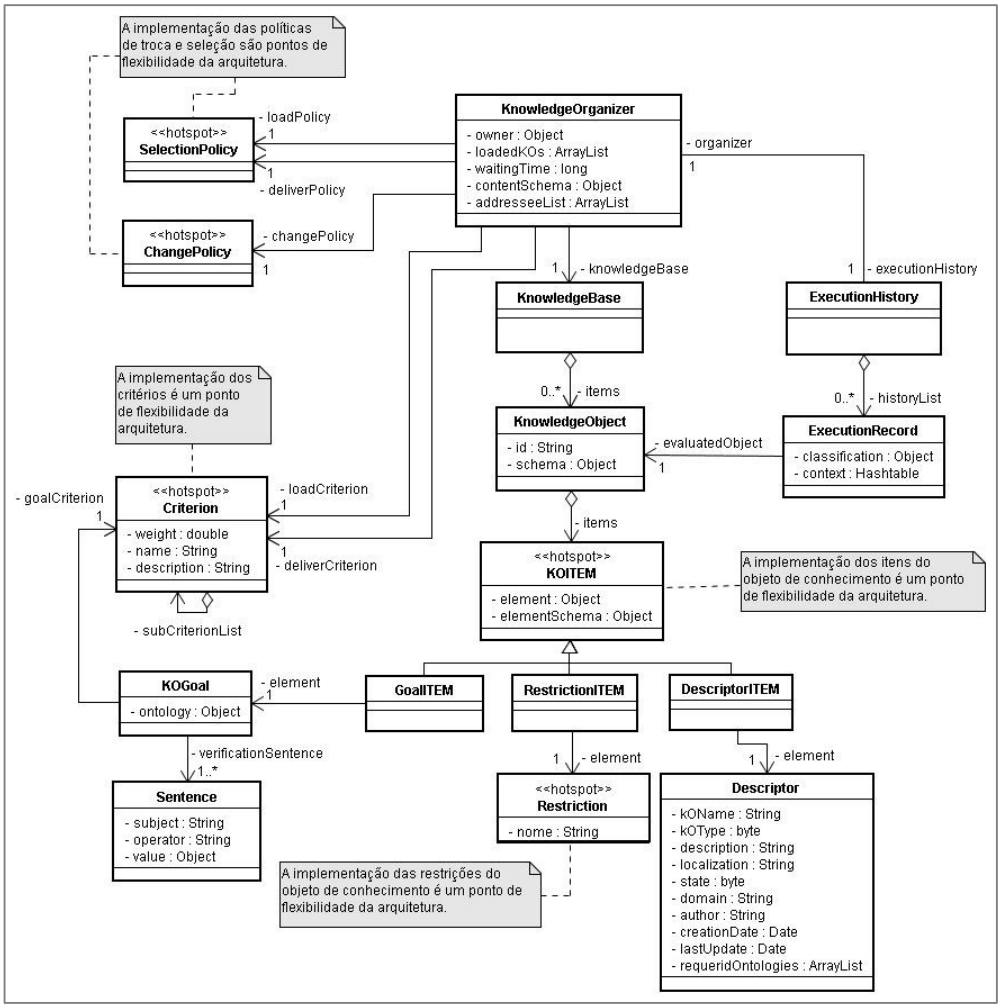


Figura 4.5: Modelo conceitual da arquitetura proposta

O segundo aspecto diz respeito ao modelo de arquitetura do agente. No *framework* proposto, trabalha-se com agentes cujos objetivos (representados em ontologias) são atingidos pela execução de um conjunto de ações, sendo que essas ações são vistas como processos de computação, ou, em outras palavras, como um conjunto de eventos produzidos e consumidos computacionalmente. Esse tipo de modelo de ações é apresentado por Ferber em (FERBER, 1999).

4.2.1 Classes e relações do modelo conceitual

A classe ***KnowledgeOrganizer*** (Organizador de Conhecimento) representa o núcleo ou classe principal da arquitetura proposta. É nessa classe que estão os principais métodos que representam as atividades do processo de GC sintetizado. Cada *KnowledgeOrganizer* possui os seguintes atributos:

- *owner* (proprietário): referência ao agente (ou parte do agente) no qual o *framework* está integrado.
- *waitingTime* (tempo de espera): tempo decorrido entre o envio de uma requisição de conhecimento e o início da análise das respostas recebidas para a solicitação.
- *loadedKOs* (objetos de conhecimento ativos): lista com os objetos de conhecimento ativos no agente. Cada objeto de conhecimento pode estar associado a apenas um objetivo do agente, ou seja, uma vez ativo, o objeto não pode ser carregado novamente enquanto não terminar a execução do objetivo ao qual já está associado.
- *contentSchema* (formato do conteúdo das mensagens): estrutura ontológica para o envio de solicitações de conhecimento e de respostas a solicitações. Esse formato permite que o conhecimento necessário ou disponível seja encapsulado, podendo ser enviado e entendido pelos agentes do sistema. Cada mensagem contém um tipo ("*requestKnowledge*" ou "*deliverKnowledge*"), um identificador (toda mensagem de solicitação tem um identificador único) e o conhecimento

requisitado ou compartilhado. Na Figura 4.6 tem-se o código OWL do formato do conteúdo das mensagens enviadas pelo *framework*.

- *addresseeList* (lista de destinatários): lista com os possíveis destinatários para as mensagens de requisição de conhecimento. Essa lista pode ser preenchida com os valores: "*all*", que indica o envio de mensagens para todos os agentes do mesmo domínio do agente solicitante (incluindo o que representa a base central de conhecimento); "*local*", que indica a verificação apenas na base de conhecimento local; "*Librarian*"[6], que é o agente que representa a base central de conhecimento (tem interfaces próprias para a inserção de objetos de conhecimento e não possui nenhum objetivo associado); ou com a identificação de qualquer outro agente (nome do agente e seu domínio). Caso a lista esteja vazia, considera-se, como padrão, o envio para todos os agentes do domínio ao qual o agente solicitante está inserido.

A classe *KnowledgeOrganizer* está associada ainda a outras cinco classes, que são: *KnowledgeBase*, *ExecutionHistory*, *Criterion*, *ChangePolicy* e *SelectionPolicy*. A base de conhecimento local (***KnowledgeBase***) nada mais é do que uma agregação de objetos de conhecimento (***KnowledgeObject***). É nessa classe que está o método responsável pela atividade armazenar do processo de GC.

O histórico de execução (classe ***ExecutionHistory***) é uma agregação de registros de execução, representados pela classe ***ExecutionRecord***. Cada registro de execução tem referência a um objeto de conhecimento (atributo *evaluatedObject*), ao resultado da execução desse objeto (atributo *classification*) e aos dados que permitiram atribuir esse resultado (armazenados na tabela *context*). O resultado da execução indica se o objetivo associado ao objeto de conhecimento foi alcançado de maneira muito satisfatória, satisfatória, de maneira regular, insatisfatória ou não foi atingido. Os critérios para atribuir essa classificação estão associados com o próprio objetivo do objeto de conhecimento (classe ***KOGoal***), como será descrito no decorrer do texto.

Como pode ser observado na Figura 4.5, há duas associações entre as classes *KnowledgeOrganizer* e ***Criterion*** (um dos pontos de flexibilidade do *framework* proposto).

---

[6] Os aspectos da base central de conhecimento serão melhor descritos na Seção 5.2.4.

Ambas representam os critérios usados para verificar a similaridade entre os objetos de conhecimento disponíveis e uma determinada solicitação/necessidade. Os critérios indexados em *deliverCriterion* são usados para avaliar a similaridade entre os objetos de conhecimento armazenados na base de conhecimento local e uma solicitação de conhecimento recebida (atividade distribuir). Já os critérios contidos em *loadCriterion* são usados para avaliar os objetos de conhecimento recebidos depois de terminado o tempo de espera de uma determinada solicitação (atividade capturar).

Cada critério, além de um nome, possui os atributos descrição e peso (que representa a importância do critério na avaliação do objeto) e pode estar associado a uma lista de critérios (associação recursiva *subCriterionList*), formando uma hierarquia de critérios.

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns="http://semanticore.pucrs.br#"
xml:base="http://semanticore.pucrs.brl">
<owl:Ontology rdf:about=""/>
<owl:Class rdf:ID="Message"/>
<owl:ObjectProperty rdf:ID="content">
  <rdfs:domain rdf:resource="#Message"/>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="id">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Message"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="messageType">
  <rdfs:domain rdf:resource="#Message"/>
  <rdfs:range>
    <owl:DataRange>
      <owl:oneOf rdf:parseType="Resource">
        <rdf:rest rdf:parseType="Resource">
          <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >deliverKnowledge</rdf:first>
        <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
        </rdf:rest>
        <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >requestKnowledge</rdf:first>
      </owl:oneOf>
    </owl:DataRange>
  </rdfs:range>
</owl:DatatypeProperty>
</rdf:RDF>
```

Figura 4.6: Código OWL do conteúdo das mensagens enviadas pelo *framework*

Para selecionar um ou mais objetos de conhecimento para distribuição ou aplicação devem ser implementadas políticas de seleção (***SelectionPolicy***). A classe *KnowledgeOrganizer* tem duas associações com a classe *SelectionPolicy*: *deliverPolicy* e *loadPolicy*. Em *deliverPolicy* é indexada a política de distribuição de conhecimento. Como exemplos de política de distribuição citam-se: somente o objeto com maior pontuação será enviado; todos os objetos com pontuação maior que um determinado valor serão enviados, etc. Já em *loadPolicy* está indexada a política de seleção de objetos de conhecimento para aplicação. Como exemplos de política de aplicação citam-se: aplicar o objeto com a maior similaridade, aplicar o objeto com a maior similaridade e o menor risco, onde risco refere-se à aplicabilidade do objeto e é medido de acordo com a abrangência do conhecimento contido no objeto, entre outros.

Com os registros de execução presentes no histórico de execução podem-se desenvolver políticas para a solicitação de conhecimento novo para atender a um determinado objetivo, o que está representado no modelo pela associação entre *KnowledgeOrganizer* e ***ChangePolicy***. A política de troca (*ChangePolicy*) está altamente relacionada ao tipo de dados utilizado para classificar o resultado das execuções. Se o resultado da execução for classificado em uma escala numérica (números de 1 a 5), por exemplo, pode-se implementar uma política que indique a necessidade de novos conhecimentos a cada 5 execuções de um determinado objeto com classificação menor do que 3 (a nota três, representa, neste caso, uma classificação pouco satisfatória). Cada vez que é armazenado um novo registro de execução, deve ser verificada a política de troca.

Um objeto de conhecimento (*KnowledgeObject*), como já mencionado, encapsula todo o conhecimento necessário para que um agente possa executar ações de forma a atingir um objetivo. A estrutura dos objetos de conhecimento, assim como a dos casos da técnica de RBC, possui duas partes básicas, que são: (1) a descrição do problema, que nos objetos de conhecimento é feita através da ontologia contida no item *KOGoal*; (2) a descrição da solução, que é feita através dos outros itens contidos no objeto.

O conhecimento encapsulado em um objeto de conhecimento deve estar vinculado aos diferentes elementos que compõem o agente ao qual o *framework* está integrado. Como cada plataforma estabelece uma arquitetura interna diferente para os seus agentes, cada objeto de conhecimento é composto de uma série de itens (***KOItem***) que são pontos de flexibilidade do *framework*. Cada *KOItem* possui dois atributos: *element*, que representa o elemento da estrutura do agente; e *elementSchema*, que é uma estrutura

ontológica com todas as informações necessárias para se recriar o item em um outro contexto (outro agente). No momento da transferência de conhecimento, a estrutura ontológica (*elementSchema*) é instanciada com as informações capturadas no item indexado pelo atributo *element* e apenas o modelo com as instâncias é transmitido.

Cada objeto de conhecimento possui, no entanto, três itens padrão, que são: *GoalITEM, DescriptorITEM* e *RestrictionITEM*. *DescriptorITEM* e *RestrictionITEM* são itens informativos. Eles trazem informações sobre o objeto de conhecimento (essas informações podem ser consideradas meta-dados e estão representadas no modelo pelos atributos da classe ***Descriptor***) e também sobre suas restrições de compartilhamento (que devem ser verificadas na atividade distribuir do processo de GC). Para a seleção dos meta-dados foi utilizado o material desenvolvido pela iniciativa *Dublin Core Metadata*[7], que é uma organização que se dedica a promover a adoção de padrões de meta-dados interoperáveis para a descrição de recursos. As restrições de compartilhamento do objeto, representadas no modelo pela classe ***Restriction***¸ são também um ponto de flexibilidade da arquitetura.

O item *GoalITEM*, por sua vez, tem como elemento o objetivo geral do objeto de conhecimento, o ***KOGoal***. No *KOGoal*, mais especificamente no seu atributo *ontology*, é indicado para qual tipo de problema que o conhecimento encapsulado no objeto está voltado. É através da ontologia indexada por *ontology* que são feitas as verificações de similaridade entre o objeto e as solicitações recebidas. *KOGoal* também está associado a uma ou mais sentenças (classe *Sentence*), através das quais é feita a avaliação da execução do objeto de conhecimento. Para fazer essa avaliação, *KOGoal* está associado à classe *Criterion*, que, por ser um ponto de flexibilidade do sistema, permite que as sentenças e os dados resultantes da execução (presentes na tabela *context*) sejam avaliados e comparados das mais diferentes formas. Cada sentença possui três atributos: (1) *subject*, que representa o nome de uma variável que deve estar contida na tabela *context*; (2) *operator*, que indica o tipo de operador utilizado (pode ser um operador relacional ou de igualdade); e (3) *value*, que é o valor requerido para a variável.

A classe *KnowledgeObject* possui ainda dois atributos, um identificador (id) e um *schema*. O *schema* (cujo tipo deve ser algum objeto que represente uma ontologia) é utilizado para representar o objeto de conhecimento no momento de seu compartilhamento.

---

[7] www.dublincore.org/

Para exemplificar os demais possíveis itens de um objeto de conhecimento para agentes de software, foi feita uma análise na literatura e nas especificações das plataformas para desenvolvimento de SMAs disponíveis, a fim de identificar um conjunto de aspectos freqüentemente encontrados nessas entidades. Como resultado deste estudo, os seguintes aspectos[8] foram identificados:

- Ações: é através de ações que se pode alterar o estado do agente ou do ambiente no qual o agente está inserido.
- Regras: é a partir de regras que o agente processa os fatos recebidos e toma decisões.
- Sensores: é através dos sensores que os agentes percebem o ambiente (eventos que ocorrem no ambiente). Cada sensor reconhece um tipo de padrão que deve ser verificado toda vez que um evento ocorre no ambiente.
- Fatos: são recursos publicados no ambiente ou que fazem referência à estrutura interna do agente. Os fatos servem direta ou indiretamente para a tomada de decisões.
- Efetuadores (*Effectors*): é através dos efetuadores que o agente age sobre o ambiente, podendo assim interagir com os outros agentes do sistema.

A integração do *framework* proposto sobre plataformas de desenvolvimento de SMAs se dá através do mapeamento e implementação dos pontos de flexibilidade dependentes das estruturas da plataforma, o que será mostrado na próxima seção.

4.3 Mapeando a arquitetura proposta sobre plataformas para o desenvolvimento de SMAs

O mapeamento do *framework* em uma plataforma para o desenvolvimento de SMAs é feito em duas grandes atividades: (1) verificação dos canais de comunicação possíveis entre o *framework* e os demais componentes do agente; (2) identificação e

[8] Acredita-se que, mesmo que não haja algum desses itens explicitamente na plataforma, possivelmente há elementos com uma semântica semelhante.

instanciação dos itens do objeto de conhecimento, onde devem ser verificados quais aspectos dos agentes que devem ser encapsulados nos objetos de conhecimento e também como esses aspectos podem ser carregados no momento da aplicação do conhecimento.

A seguir, para demonstrar a integração do *framework* em plataformas para o desenvolvimento de SMAs, será mostrado o mapeamento sobre duas plataformas.

4.3.1 JADE

Para a integração do *framework* proposto em agentes JADE, faz-se necessário a criação de um novo atributo na classe *jade.core.Agent* (todo agente JADE é uma classe que herda da classe *jade.core.Agent*) para conter uma referência a classe principal do *framework* proposto, a *KnowledgeOrganizer*. A configuração dos parâmetros do *framework* pode ser feita diretamente no método *setup()* da classe *jade.core.Agent*, que é implementado para as inicializações do agente. Nesse método podem ser especificadas, por exemplo, as políticas de troca e de seleção, os critérios para a avaliação dos objetos de conhecimento e também pode ser feito o cadastramento de objetos de conhecimento na base de conhecimento do agente.

No JADE, as tarefas executadas por um agente são representadas por comportamentos implementados como objetos de uma subclasse de *jade.core.behaviours.Behaviour*. O método *addBehaviour()* (da classe *Agent*) permite que o agente execute uma tarefa definida por um objeto de comportamento. A classe que herda de *Behaviour* precisa implementar o método *action()*, para definir as operações presentes na execução do comportamento, e o método *done()*, para especificar o término de um comportamento e sua remoção do *pool* de comportamentos.

Agentes JADE, no entanto, não possuem nenhum conjunto de objetivos representados semanticamente (através de ontologias). A execução de suas tarefas está relacionada apenas ao conteúdo do método *action()* da classe *Behaviour* e não há qualquer representação semântica do que é realizado em um determinado comportamento. Assim, para integrar o *framework* proposto em agentes JADE, ao invés de uma lista de *behaviours*, os agentes devem estar relacionados a uma lista de objetivos que, além de uma associação com um *behaviour* que permita sua execução, devem possuir um atributo do tipo ontologia para

representar o conhecimento manipulado no *behaviour*. A criação da lista de objetivos e sua associação com os *behaviours* deve ser implementada para que se possa fazer a integração.

Também, para que seja iniciado o processo de aquisição de novos conhecimentos para determinado objetivo, o *framework* deve ser notificado quando não houver nenhum *behaviour* associado ao objetivo que deve ser executado, ou seja, um método para a notificação do *framework* deve ser implementado na classe que representa os objetivos do agente.

O classe *Behaviour*, como já mencionado, possui um método chamado *done()*, que indica quando sua execução foi terminada. Para a criação dos registros de execução do histórico de execução do *framework*, o método citado, ao invés de retornar um valor do tipo *boolean*, deverá retornar uma tabela com variáveis e seus valores finais. As variáveis a serem notificadas deverão ser inseridas a partir do construtor da classe *Behaviour*.

O paradigma de comunicação adotado pelo JADE é a passagem de mensagens assíncronas. Cada agente tem uma fila de mensagens. Quando uma mensagem enviada entra na fila de um agente, o mesmo é notificado, mas o processo de tratamento da mensagem é de responsabilidade do programador. Assim, para que o *framework* receba as mensagens com as requisições de conhecimento vindas de outros agentes, ele deve ser notificado a cada nova mensagem que chega na fila de mensagens, para que, se necessário, seja iniciado o processo de distribuição de conhecimento. Para diminuir o número de mensagens verificadas no *framework*, pode-se utilizar *templates* para retirar mensagens específicas da fila de mensagens do agente (esses *templates* são implementados no JADE como instâncias da classe *jade.lang.acl.MessageTemplate*).

Para o envio de mensagens no JADE, é necessário preencher os campos de um objeto *ACLMessage* e chamar o método *send()* da classe *Agent*. Como a classe *KnowledgeOrganizer* do *framework* proposto terá uma referência ao agente (seu "proprietário"), o envio de mensagens de requisição ou de distribuição de conhecimento também será feito através da criação de um objeto *ACLMessage* (com o conhecimento necessário no campo *content*) e da chamada do método *send()* do agente proprietário.

No JADE, cada objeto de conhecimento será composto por dois tipos de itens (além dos *default* do *framework*): *BehaviourITEM* e *RuleITEM*. O *BehaviourITEM* terá como elemento um objeto da classe *Behaviour*, que deve conter todas as ações necessárias para que o agente atinja o objetivo do objeto de conhecimento ao qual está associado. Cada objeto de

conhecimento deve possuir apenas um elemento do tipo do *BehaviourITEM* (o que não impede que o *behaviour* indexado pelo atributo *element* possua outros "*sub-behaviours*"). Esta restrição de cardinalidade está relacionada à aplicação do objeto de conhecimento. No momento em que o objeto for aplicado, o *behaviour* contido nos itens do objeto será associado ao objetivo do agente com conhecimento faltante e, por definição, cada objetivo deve estar associado a apenas um *behaviour* por vez.

O item *RuleITEM* terá como elemento as regras utilizadas pelo agente para raciocinar sobre os fatos do mundo. O JADE é integrado ao JESS - *Java Expert System Shell* (JESS, 2006), que é uma ferramenta baseada em regras para raciocínio simbólico.

Para a criação dos objetos que representam as ontologias utilizadas no *framework* sugere-se o uso da classe *Ontology*, que é uma classe do próprio JADE utilizada para a criação de vocabulários compartilhados entre os agentes.

4.3.2 SemantiCore

No SemantiCore, o *framework* proposto será integrado como um novo componente, chamado componente **Organizador de Conhecimento ou Organizador**. A ligação entre os componentes do SemantiCore 2006 (ou caminho de dados) é um ponto de flexibilidade da arquitetura, o que facilita a integração. Como pode ser visto na Figura 4.7, que mostra a comunicação entre os componentes, o Organizador se comunicará com o componente sensorial (de onde virão as mensagens recebidas pelo ambiente), com o decisório (de onde virão mensagens de solicitação de conhecimento do próprio agente), com o efetuador (para solicitar o envio de mensagens pelo ambiente) e com o executor (de onde virão as notificações de término de execução).

Os agentes no SemantiCore, assim como o descrito na Seção 2.3.4.1, possuem uma estrutura orientada a componentes e, em cada um dos componentes pode-se identificar uma entidade ou parte principal, que caracteriza o seu funcionamento. O componente sensorial, por exemplo, é caracterizado pelo seu conjunto de sensores, através dos quais os agentes percebem o ambiente. Já o componente decisório, armazena uma série de fatos e de regras de inferência para a tomada de decisão. O componente executor, por sua vez, é

caracterizado pelas ações que encapsula. Por último, o componente efetuador é composto por efetuadores, que representam cada um dos tipos possíveis de mensagens transmitidas pelo ambiente. Para representar todos esses elementos, devem ser criados, respectivamente, cinco tipos de itens no objeto de conhecimento: *SensorITEM, RuleITEM*, *ActionPlanITEM*, *FactITEM* e *EffectorITEM*. As linhas tracejadas na Figura 4.7 indicam os componentes que o Organizador manipulará para aplicar um novo objeto de conhecimento ou, em outras palavras, para carregar os itens do objeto de conhecimento.

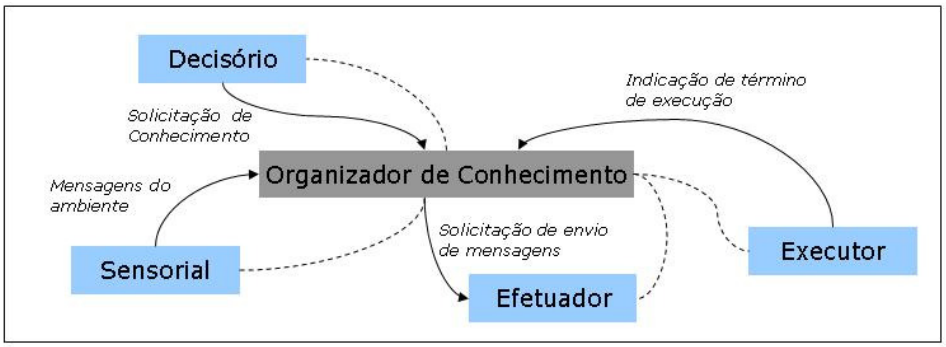


Figura 4.7: Comunicação entre o Organizador de Conhecimento e os demais componentes do SemantiCore

Cada objeto de conhecimento pode estar associado a apenas um item que tenha como elemento um plano de ação. Essa restrição, assim como explicado na seção anterior, está envolvida com a aplicação do objeto de conhecimento (no SemantiCore, o item que ficará relacionado com o objetivo do agente é o plano de ação). No Capítulo 7 serão feitas mais considerações sobre essa restrição.

O objetivo do objeto de conhecimento ficará no próprio Organizador, que deverá verificar o resultado de execução do objeto (se satisfatória ou não) através dos critérios associados ao seu objetivo. A coleta de dados para a realização desta avaliação é feita verificando-se o conteúdo das variáveis do plano de ação ao final de sua execução.

Para a definição e o uso de ontologias no SemantiCore será usado o *framework* Jena. No SemantiCore, embora o mecanismo decisório seja um ponto de flexibilidade da arquitetura, já há uma integração nativa com o Jena, o que possibilita o uso de máquinas de inferência neste componente e a criação ou manipulação de ontologias nos demais componentes do agente.

4.4 Considerações sobre o capítulo

Este capítulo apresentou os aspectos gerais do modelo de arquitetura para Gestão de Conhecimento desenvolvido e aplicado nesta dissertação. Na descrição, procurou-se destacar os aspectos conceituais da arquitetura proposta, como a aplicação de cada atividade do processo de GC sintetizado em uma arquitetura de software. Também, foram salientadas questões referentes à integração do *framework* proposto em plataformas para o desenvolvimento de SMAs, onde foi apresentado um breve estudo sobre a integração da arquitetura no JADE e no SemantiCore.

Na Tabela 4.1, tem-se um resumo com os principais aspectos que devem ser observados para a integração do *framework*. Como pode ser observado na tabela, devem ser verificadas questões referentes à comunicação, aos itens dos objetos de conhecimento, aos objetivos do agente e às ferramentas para criação e edição de ontologias. Para cada aspecto apresentado na tabela, há uma descrição indicando a implementação esperada. Por exemplo, para o recebimento de mensagens do ambiente, acredita-se que serão criados canais de comunicação entre o agente e o *framework*.

No próximo capítulo serão detalhados os aspectos referentes à implementação do *framework* e à sua integração ao SemantiCore. A implementação de todos os aspectos ilustrados na Tabela 4.1 será apresentada de forma dispersa ao longo do todo o capítulo.

Tabela 4.1: Aspectos que devem ser considerados para a integração do *framework*

	Aspecto a ser verificado	Descrição
Comunicação	Recebimento de mensagens	Para que o *framework* seja notificado das mensagens de solicitação e de distribuição de conhecimento recebidas pelo agente através do ambiente, deve ser criado um canal de comunicação entre o agente e o *framework*.
	Envio de mensagens	Para que o *framework* possa enviar mensagens de solicitação e de distribuição de conhecimento, ele deve ter acesso às estruturas utilizadas nas mensagens da arquitetura e também aos métodos que permitem que as mensagens sejam enviadas pelo ambiente.
Objeto de Conhecimento	Identificação dos itens	Identificação dos aspectos da arquitetura do agente que devem compor um objeto de conhecimento. Para tanto, deve-se verificar quais são os elementos necessários para a execução de um objetivo do agente.
	Criação da estrutura ontológica dos itens	Compreende o levantamento das informações necessárias para a recriação dos elementos do agente indexados nos itens do objeto de conhecimento. Essas informações devem estar explicitadas em uma estrutura ontológica.
	Carga e remoção dos itens	Durante a aplicação de um objeto de conhecimento, cada um de seus itens deve ser carregado no agente. Para tanto, após a recriação da classe que representa o item, devem existir métodos que indiquem como o item é relacionado à arquitetura do agente. A remoção indica que, assim como os itens podem ser carregados no agente, deve ser possível removê-los quando o objeto de conhecimento não se demonstrar mais útil.
Objetivos dos agentes	Criação de uma estrutura para representar os objetivos	No *framework*, acredita-se que o agente possui uma série de objetivos (que são representados através de ontologias) e que esses objetivos estão associados a algum elemento da arquitetura dos agentes que permita a sua execução. Caso não haja este elemento na arquitetura, ele deve ser criado.
	Identificação do elemento que deve ser relacionado ao objetivo	Compreende a identificação do elemento que ficará associado aos objetivos do agente (caso isto já não esteja definido na arquitetura). É através da execução deste elemento que o objetivo é satisfeito.
	Indicação de conhecimento faltante	Para que o *framework* seja notificado sobre a falta de conhecimento no agente, o agente deve ser capaz de perceber uma necessidade de conhecimento e sinalizá-la ao *framework*. Para tanto, sugere-se que seja verificada a existência ou não da associação dos objetivos com os elementos para as suas execuções. Caso não haja a associação quando o objetivo for iniciado, o *framework* deve ser notificado.
	Aviso de término da execução	Para que seja computada a classificação da execução do objetivo, o *framework* deve ser notificado ao término da execução do elemento associado ao objetivo do agente. Essa notificação deve conter o contexto final da execução do elemento, através do qual será realizada a verificação da classificação.
Ontologia	API para criação de ontologias	Para a criação e edição das ontologias, deve ser utilizada alguma API para a manipulação de ontologias. No *framework*, há uma integração nativa com o Jena.

5 PROTÓTIPO E IMPLEMENTAÇÃO

5.1 Introdução

Neste capítulo, a arquitetura será detalhada em termos de sua implementação. Para o detalhamento, procurou-se seguir a mesma estrutura de apresentação do modelo conceitual (Capítulo 4), visando uma melhor identificação entre a parte conceitual e a parte prática.

A implementação do *framework* foi feita tendo-se como base o SemantiCore (na sua versão 2006), ou seja, durante a implementação do *framework* já foram considerados e implementados os aspectos de integração com esta plataforma de desenvolvimento de SMAs. O aproveitamento de toda a infra-estrutura existente no SemantiCore teve como objetivo a diminuição do tempo de implementação. Para a manipulação e criação de ontologias, como já citado, foi utilizada a API do Jena.

Durante a implementação procurou-se observar todos os aspectos com constantes modificações para que eles pudessem ser implementados em estruturas flexíveis e expansíveis. Este é o caso dos critérios de verificação, que podem ser usados tanto para avaliar a similaridade entre as solicitações de conhecimento recebidas e os objetos de conhecimento disponíveis quanto para avaliar a classificação da execução de um objeto.

Quanto ao escopo da implementação, neste primeiro momento estão sendo implementados apenas os aspectos referentes a dois dos fluxos do processo de GC: o Fluxo 2, que trata do compartilhamento do conhecimento; e o Fluxo 3, que considera a aquisição e aplicação de novos conhecimentos. O Fluxo 1, que envolve a criação e representação de novos conhecimentos, está sendo executado apenas de forma manual, ou seja, a criação dos objetos de conhecimento é feita pelo desenvolvedor no momento da configuração do agente. As razões para a delimitação do escopo da implementação estão relacionadas às dificuldades em se identificar conhecimento novo no agente e também na incerteza quanto à relevância e reusabilidade do conhecimento novo identificado. Esses aspectos serão melhor discutidos no Capítulo 7, que traz as considerações finais e os trabalhos futuros.

Por último, acredita-se que algumas reformulações da versão inicial serão necessárias para o aperfeiçoamento tanto do desempenho quanto de outras questões identificadas a partir da primeira versão. É improvável que se possa chegar a uma arquitetura ideal em flexibilidade e desempenho que combine várias tecnologias na sua primeira versão.

5.2 O Componente Organizador de Conhecimento

Para descrever os aspectos relacionados à implementação, na Figura 5.1 está ilustrada a estrutura do *framework* proposto já integrada ao SemantiCore. Como os detalhes da implementação serão apresentados considerando-se a integração ao SemantiCore, a classe principal do *framework* (*KnowledgeOrganizer)* passou a ser denominada *KnowledgeOrganizerComponent*, visto que no SemantiCore o *framework* será integrado como um novo componente (herdando os aspectos da classe *semanticore.domain.model.Component*). Por questões referentes ao tamanho do modelo, na Figura 5.1 não estão representados os métodos do tipo *get()* e *set()*.

Para facilitar o entendimento, os aspectos da implementação serão apresentados através de quatro seções: (1) aspectos gerais, onde serão descritas as principais classes que constituem o *framework* e também seus métodos; (2) itens do objeto de conhecimento, onde será apresentado cada item que compõe um objeto de conhecimento no SemantiCore, salientado-se questões referentes a serialização e a recriação, a partir das ontologias, dos elementos indexados; (3) adaptações requeridas no SemantiCore, onde serão explorados os códigos acrescentados a algumas classes do SemantiCore; (4) base de conhecimento global, onde serão explicitados os aspectos referentes à criação desse elemento da arquitetura.

5.2.1 Aspectos gerais

Como pode ser visto na Figura 5.1, a classe *KnowledgeOrganizerComponent* encapsula as funções necessárias para a busca, a distribuição e a aplicação de conhecimento

(objetos de conhecimento). Essas funções são acionadas de acordo com as necessidades do agente ou com o recebimento de requisições de outros agentes. As necessidades de conhecimento são verificadas no *framework* após suas sinalizações pelo método *put()*. Por exemplo, considere que um agente, para atingir determinado objetivo, precisa adquirir conhecimento sobre vender livros. Esta necessidade será indicada ao *framework* que então executará o método *captureKnowledge()* para a busca de objetos de conhecimento que tratem da venda de produtos. Após o recebimento dos objetos de conhecimento e da seleção de um deles, é a vez de executar o método *loadKnowledge()*.

Cada vez que é recebida uma solicitação de conhecimento do agente ao qual o *framework* está integrado é criado um objeto do tipo *KnowledgeRequest*. Cada *KnowledgeRequest* possui quatro atributos, que são: *id*, que é o identificador gerado para a requisição (deve ser um identificador único); *startTime*, que informa o horário em que foi solicitado o conhecimento (necessário para a verificação do tempo de espera); *knowledgeRequested*, que é uma ontologia com a representação do conhecimento faltante (é essa ontologia que é enviada na mensagem de solicitação de conhecimento); e *receivedKnowledge*, que é uma lista com todas as mensagens de distribuição de conhecimento recebidas para a solicitação.

As solicitações de conhecimento do agente são gerenciadas pela *thread* *RequestVerifier*. Enquanto há solicitações de conhecimento pendentes na lista de solicitações (*requestList*) é verificado o tempo decorrido de cada solicitação (através do método *verifyTimeOver()* da classe *KnowledgeRequest*). Cada vez que o *framework* é sinalizado com uma mensagem cujo tipo é "*deliverKnowledge*", essa mensagem é encaminhada ao objeto *requestVerifier* através do método *notifyReceivedKnowledge()*. Quando chega uma mensagem de distribuição de conhecimento em *RequestVerifier*, primeiro é verificado o identificador da solicitação ao qual o conhecimento distribuído está relacionado. Se a solicitação ainda está pendente (não acabou o tempo de espera), a mensagem é adicionada à lista de conhecimentos recebidos da solicitação. No momento que termina o tempo de espera para uma solicitação, é executado o método *verifyReceivedKnowledgeForRequest()*, de *RequestVerifier*. Neste método é verificado todo o conhecimento recebido para a solicitação (com o uso dos critérios de aplicação) e é selecionado um objeto para o carregamento, de acordo com a política de aplicação referenciada na classe *KnowledgeOrganizerComponent*.

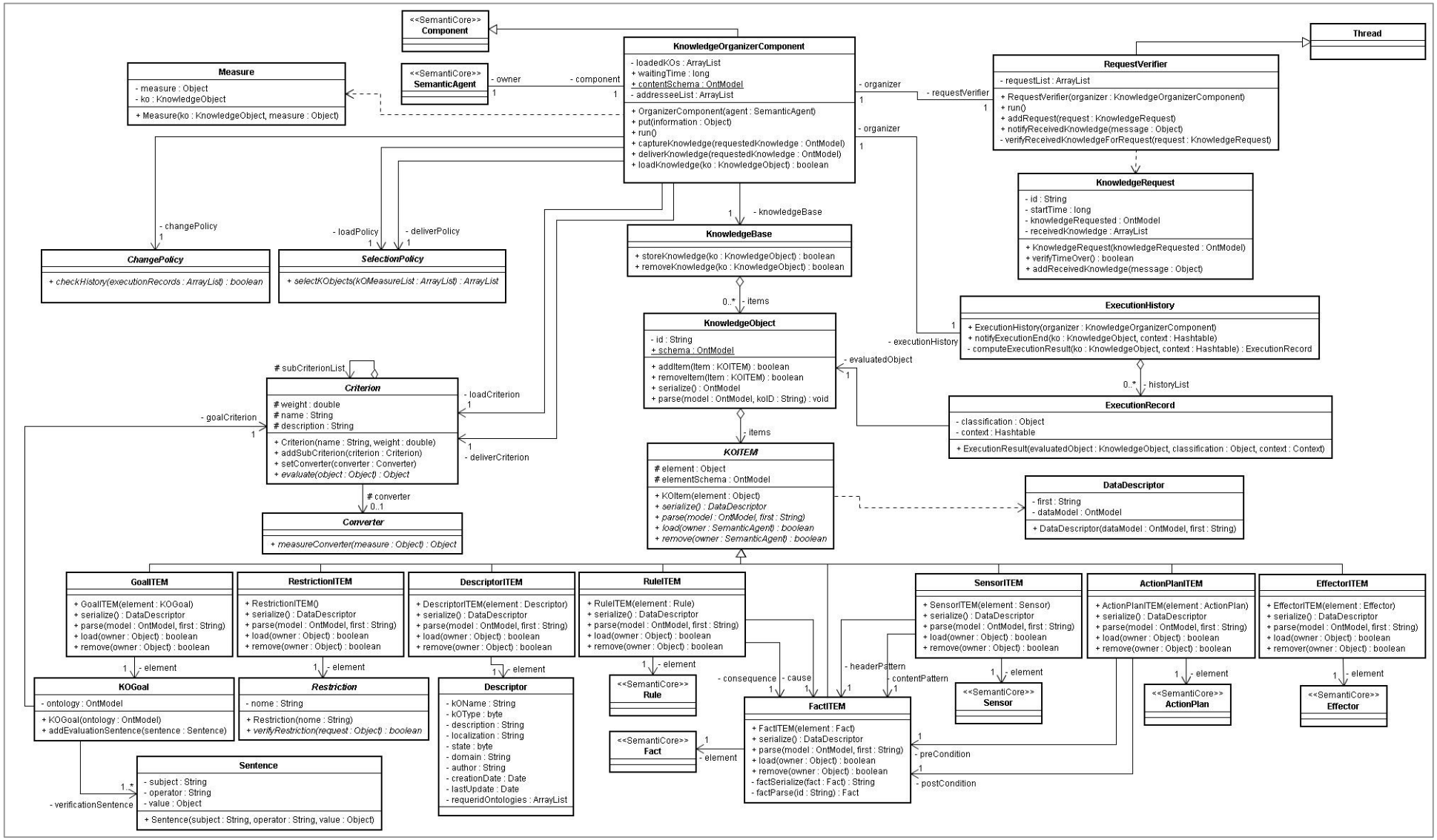


Figura 5.1: Implementação do componente Organizador de Conhecimento

Já as solicitações de conhecimento de outros agentes são processadas no método *deliverKnowledge()* (da classe *KnowledgeOrganizerComponent*). Cada vez que é recebida uma solicitação, cujo conhecimento requerido é estruturado na forma de uma ontologia, é feita uma verificação de similaridade entre a solicitação e todos os objetos de conhecimento disponíveis na base de conhecimento. Como resultado desta verificação tem-se uma lista de objetos da classe *Measure* (que possui uma referência ao objeto de conhecimento e outra a "nota" atribuída a ele). Essa lista é então encaminhada ao método *selectKObjecs()* da política de seleção para distribuição (*deliverPolicy*) relacionada à classe *KnowledgeOrganizerComponent*. A implementação do método *selectKObjecs()* é também um ponto de flexibilidade da arquitetura e deve retornar uma lista com os objetos aptos para o uso, neste caso, aptos para o envio ao agente solicitante.

Na base de conhecimento local, que possui uma lista com os objetos de conhecimento disponíveis no agente, estão os métodos para o armazenamento e a remoção de objetos de conhecimento. Esses métodos são invocados, normalmente, por métodos da classe *KnowledgeOrganizerComponent* ou na inicialização do agente, onde podem ser adicionados objetos na base.

O *framework* estrutura o conhecimento de um agente na forma de objetos de conhecimento. Cada objeto de conhecimento possui um nome, uma estrutura ontológica usada no seu compartilhamento (atributo *schema* da classe *KnowledgeObject*) e uma série de itens (classe *KOITEM*). A ontologia[9] contida no *schema* da classe *KnowledgeObject*, assim como mostra a Figura 4.6, possui duas classes OWL, *KnowledgeObject* e *KOItem*. Essas classes se relacionam através da *object property hasItems*. Cada *KOItem* possui outras duas propriedades: *itemClass*, que é uma *datatype property* cujo valor é uma *string*; e *element*, que é uma *object property* cujo alvo pode ser de qualquer tipo (como pode ser visto na figura, tem-se como alvo uma *owlClass* genérica).

Quando o objeto de conhecimento é serializado para ser transmitido, cada um de seus itens é também serializado e as ontologias resultantes destes processos são adicionadas a ontologia do objeto de conhecimento. A integração entre os indivíduos das ontologias vindas dos itens e os presentes na ontologia do objeto de conhecimento é feita pela *object property element* (*owlClass KOITEM*) que, devido a os itens serem pontos de flexibilidade da arquitetura, pode ser de qualquer tipo. Exemplos da criação de objetos de

[9] A ontologia está representada em notação UML, conforme o especificado em (OMG, 2006).

conhecimento e do processo de instanciação da ontologia do objeto serão mostrados no Capítulo 6, que traz o desenvolvimento de uma aplicação.

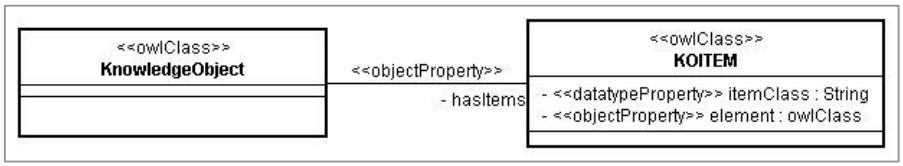


Figura 5.2: Representação ontológica dos objetos de conhecimento

Avaliar a eficiência do conhecimento utilizado em determinada tarefa indica que o sistema deve monitorar o seu próprio desempenho e ser capaz de aprender com isso. Neste sentido, o *framework* possui um histórico de execução (*ExecutionHistory*) composto por vários registros de execução. A capacidade de monitorar seu desempenho, de acordo com Weber e Wu (2004), pode garantir longos períodos de execução ao sistema sem a necessidade de manutenção.

Cada vez que é sinalizado o término da execução de um objetivo do agente (término do plano de ação associado ao objetivo) no método *put()* de *KnowledgeOrganizerComponent*, deve ser verificado se há algum objeto de conhecimento ativo no agente (lista *loadedKOs*) relacionado ao objetivo finalizado. Caso afirmativo, é chamado o método *notifyExecutionEnd()* da classe *ExecutionHistory*. Na chamada deste método, além da indicação do objeto de conhecimento cuja execução foi finalizada, deve ser indicado o *contexto* referente ao final da execução do plano. O palavra *contexto* vem da arquitetura do SemantiCore e indica uma lista de variáveis e seus respectivos valores. No SemantiCore, as variáveis do contexto e os seus valores podem tanto ser adicionados no momento da criação do plano de ação quanto em tempo de execução (por exemplo, dentro do método *exec()* de uma das ações do plano). As informações contidas no contexto podem ser recuperadas e modificadas por qualquer ação do plano.

A partir das informações contidas no contexto, pode-se classificar a execução do objeto de conhecimento através da execução do método *computeResult()* da classe *ExecutionHistory*. Essa classificação é feita de acordo com as sentenças de verificação associadas à classe *KOGoal*, que está no *GoalITEM* do objeto de conhecimento. As sentenças contêm o nome da variável, um operador (“>”, “<’, “=”) e um valor. A comparação dos valores contidos na tabela de contexto com os requeridos pelas sentenças associadas ao objetivo do objeto de conhecimento é feita através de algum critério, que deve ser implementado pelo desenvolvedor (os critérios são pontos de flexibilidade da arquitetura). Por

exemplo, se um objetivo tem uma sentença do tipo (“preço”, “<”, “50”) e, ao término da execução do plano de ação o contexto do plano possui o valor “40” para a variável “preço”, isso indica que a sentença foi satisfeita, visto que o preço ficou abaixo do nível máximo estabelecido. À classificação do objetivo é atribuído o valor que retorna do método *evaluate()* (classe *Criterion*). Para padronizar esse valor, pode ser usado um objeto da classe *Converter*, que converte uma medida de entrada em algum outro valor (por exemplo, um valor 70, que indica uma porcentagem de 70% de satisfação, pode ser convertido, em uma escala lingüística, para o valor “bom”).

Depois de computada a classificação da execução de um objeto de conhecimento, é criado um novo registro de execução e inicia-se o processo de avaliação de seu histórico de execução. Nesse processo, todos os registros de execução do objeto finalizado são enviados para o método *checkHistory()* da classe *ChangePolicy*, que também é um ponto de flexibilidade da arquitetura. Dentro deste método deve ser avaliado o histórico de execução para verificar a necessidade de substituição do conhecimento. Como resultado desta avaliação é retornado o valor verdadeiro, que dispara o processo de captura de novos conhecimentos, ou falso, que não dispara nenhuma ação.

A Tabela 5.1 apresenta todos os possíveis eventos que são sinalizados ao *framework* (através do método *put()* de sua classe principal – *KnowledgeOrganizerComponent*) e quais as ações que eles acarretam (método que é disparado com a chegada de cada evento).

Tabela 5.1: Eventos sinalizados ao *framework*

Possíveis sinalizações	Método disparado – Classe	Observações
Requisição de conhecimento interna	*captureKnowledge() - KnowledgeOrganizerComponent*	A partir do método indicado é criado um objeto do tipo *KnowledgeRequest* e é feita uma solicitação aos outros agentes do sistema.
Requisição de conhecimento externa	*deliverKnowledge() - KnowledgeOrganizerComponent*	Cada vez que é recebida uma mensagem de solicitação de conhecimento de outros agentes é iniciado o processo de distribuição de conhecimento, onde todos os objetos de conhecimento disponíveis são avaliados e os mais compatíveis são enviados ao agente solicitante.
Indicação do término de uma execução	*notifyExecutionEnd() - ExecutionHistory*	Quando é indicado o término da execução de um plano, um novo objeto do tipo *ExecutionRecord* é criado.
Resposta à solicitação de conhecimento	*notifyReceivedKnowledge() - RequestVerifier*	Toda resposta recebida para uma solicitação é armazenada em sua lista de respostas para posterior avaliação.

5.2.2 Itens do objeto de conhecimento

Um objeto de conhecimento é uma agregação de itens, representados pela classe *KOITEM* no modelo. Cada *KOITEM* possui quatro métodos não implementados (pontos de flexibilidade da arquitetura), que são: *serialize()*, *parse()*, *load()* e *remove()*. Os dois primeiros métodos estão relacionados com o compartilhamento do objeto de conhecimento, onde cada item deve poder ser serializado para transferência e também ser reconstituído no destino. O método *serialize()* retorna um objeto da classe *DataDescriptor*. Cada *DataDescriptor* contém uma ontologia com as instâncias do item serializado e uma *string* indicando qual a instância que deve ser diretamente relacionada com o indivíduo que representa o objeto de conhecimento. Como na ontologia que representa o item podem-se ter várias classes e, conseqüentemente, várias instâncias de classes diferentes, tornou-se necessário indicar qual o elemento "base" ou principal da ontologia de dados, de onde se podem derivar os demais relacionamentos.

Nos métodos *load()* e *remove()* é passado como parâmetro um objeto da classe *SemanticAgent*, que representa o "proprietário" do *framework*. No SemantiCore, todos os agentes do sistema herdam as propriedades desta classe e através de seus métodos é possível acessar qualquer um dos componentes do agente, o que é necessário na carga de alguns dos itens do objeto de conhecimento.

Todo objeto de conhecimento, independentemente da plataforma de integração, possui três itens padrão (*GoalITEM*, *RestrictionITEM* e *DescriptorITEM*), como já mencionado no capítulo anterior. Todos esses itens possuem os quatro métodos abstratos herdados da classe *KOITEM*, mas nenhum deles implementa os métodos *load()* e *remove()*, visto que não são aplicados, e consequentemente não precisam ser removidos, em nenhum elemento do agente em que o *framework* está integrado. O tratamento destes itens é feito no próprio *framework* e eles permanecem ligados apenas ao objeto de conhecimento do qual fazem parte.

O *GoalITEM*, que contém como elemento a ontologia que define o conhecimento armazenado no objeto de conhecimento, possui o *elementSchema* apresentado na Figura 5.3. Como pode ser visualizado na figura, há três classes OWL: *GoalITEM*, *Criterion* e *Sentence*. Na classe OWL *GoalITEM*, há uma *datatype property* denominada *hasOntology* cujo valor alvo é uma *string*. É nesta propriedade que é inserida a ontologia

indexada no atributo *element* da classe *GoalITEM*. A ontologia do objetivo é inserida nesta estrutura ontológica como uma *string* para possibilitar a sua recriação quando é feito o *parse* do objeto de conhecimento (para não misturar os conceitos do objetivo com os dos itens do objeto de conhecimento). A classe OWL *Sentence* é bastante similar à classe *Sentence* do modelo, onde os atributos se tornaram *datatype properties*. A classe OWL *Criterion*, no entanto, além dos atributos necessários para a recriação dos critérios (*name* e *weight*) e de sua descrição (*description*), possui duas outras *datatype properties*: *criterionClass* e *code*. Em *criterionClass* é armazenado o nome da classe que implementa o critério. Porém, se a classe não estiver disponível para instanciação, pode-se pegar o código contido na *datatype property code* para a recriação da classe que implementa o critério associado ao objetivo.

No momento da serialização do item *GoalITEM* é criado um novo indivíduo da classe OWL *GoalITEM* que é então relacionado a um indivíduo da classe OWL *Criterion* e a tantos indivíduos da classe OWL *Sentence* quantos forem os elementos da lista *verificationSentence* contida na classe *KOGoal* do modelo.

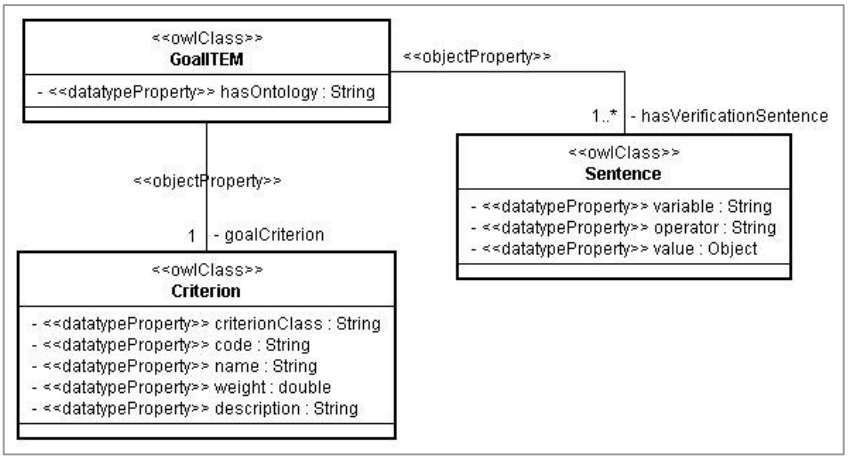


Figura 5.3: Estrutura ontológica para o *GoalITEM*

O item *RestrictionITEM*, por ser um item que contém um elemento que necessita ser implementado (o método *verifyRestriction* da classe *Restriction* é um ponto de flexibilidade da arquitetura) possui na sua estrutura ontológica apenas o nome da restrição, o nome da classe que a implementa e o código da classe (para o caso de a classe não estar disponível). O método *verifyRestriction()* tem como parâmetro a requisição de conhecimento recebida que, após ser analisada pelo código inserido pelo desenvolvedor, deve retornar um valor verdadeiro ou falso, onde verdadeiro indica que o objeto pode ser compartilhado e falso

indica que não pode. A ontologia desenvolvida para descrever as informações deste item pode ser visualizada na Figura 5.4.

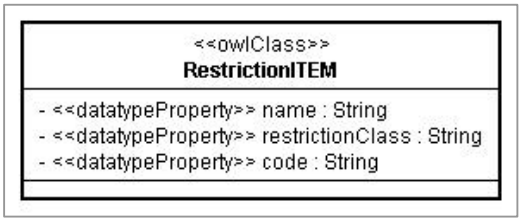


Figura 5.4: Estrutura ontológica para o *RestrictionITEM*

O item *DescriptorITEM*, por sua vez, tem como elemento um objeto da classe *Descriptor*, que traz várias informações (meta-dados) sobre o objeto de conhecimento ao qual está associado. Estas informações podem ser utilizadas, por exemplo, nos critérios para aplicação do objeto. Como exemplo deste uso, cita-se a verificação do estado do objeto (um objeto de conhecimento acabado, dependendo do caso, deve ter prioridade na aplicação sobre um objeto em construção). A estrutura ontológica para este item apresenta todos os atributos da classe *Descriptor* na forma de *datatype properties*.

5.2.2.1 Itens do objeto de conhecimento específicos do SemantiCore

Antes de descrever os aspectos de implementação dos itens que compõem um objeto de conhecimento no SemantiCore é necessário entender a semântica atribuída aos fatos (classe *semanticore.agent.kernel.information.Fact*), visto que muitos dos itens do objeto estão associados à classe *FactITEM* (que tem como elemento um objeto *Fact*). Os fatos são recursos (onde recurso representa qualquer entidade especificada em uma ontologia) que são publicados no ambiente ou que fazem referência à estrutura interna do agente como, por exemplo, a exclusão de um plano de ação. No SemantiCore, os fatos são usados como padrão de seleção de mensagens nos sensores, como pré e pós-condição das ações, e contribuem direta ou indiretamente para a tomada de decisão do agente.

A classe *Fact* possui três especializações, *ComposedFact*, *SimpleFact* e *FunctionBasedFact,* conforme ilustrado na Figura 5.5 (a). Os fatos compostos são formados por fatos simples ou compostos ou por fatos baseados em funções, ligados por um operador lógico. Os fatos simples possuem três atributos (*subject*, *predicate* e *object*) que representam

os três elementos de uma declaração RDF (*statement* RDF). Já os fatos baseados em função representam os fatos que dependem de uma avaliação de outros fatos para serem gerados. Esses fatos são usados, principalmente, nas regras de inferência. Com eles, pode-se verificar, por exemplo, se X e Y, ambos recursos RDF e literais do tipo inteiro, são iguais ("*equal* (X, Y)") ou se X é menor que Y ("*lessThan* (X, Y)").

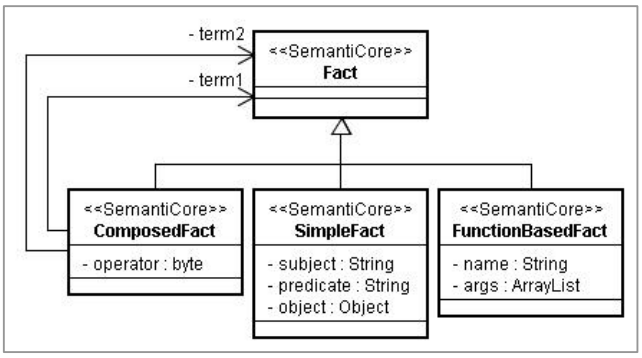
Para armazenar de forma semântica as informações necessárias para a criação dos fatos (sejam fatos simples, compostos ou baseados em função) foi definida a estrutura ontológica ilustrada na Figura 5.5 (b). Note que os diagramas mostrados na figura são bastante semelhantes, o que é justificado pela fácil geração de uma representação em OWL a partir de um esquema UML (Doan *et al.*, 2004; Cranefield, 2001).

Cada vez que é solicitada a serialização de um objeto do tipo *FactITEM*, a ontologia ilustrada na Figura 5.5 (b) é instanciada com as informações retiradas do elemento *Fact* indexado no atributo *element* de *FactITEM*. A Figura 5.6 mostra um trecho de código OWL com as informações retiradas do fato baseado em função ("*lessThan* (X, Y)"). Nesta ontologia, como pode ser visto na figura, os argumentos do referido fato são armazenados por valores do tipo *string* sendo que as *strings* começam com um número, que representa a posição do argumento na chamada da função. Isso foi necessário, pois a ordem dos argumentos pode alterar o resultado de alguns tipos de função e, embora os valores sejam inseridos na ontologia na mesma ordem em que estão na chamada do método, isto não garante a leitura seqüencial para a recriação da lista de argumentos.

Como os fatos podem estar associados a outros fatos (um objeto *ComposedFact* possui dois atributos do tipo *Fact*), foram criados dois métodos recursivos (*factParse()* e *factSerialize()*) em *FactITEM* (o que pode ser visualizado na Figura 5.1). Esses métodos são invocados, respectivamente, nos métodos herdados *parse()* e *serialize()*. Os demais métodos herdados (*load()* e *remove()*) não possuem implementação, pois, como todo *FactITEM* contido no objeto de conhecimento deverá estar relacionado a algum outro item do objeto, estes processos são realizados juntamente com o elemento ao qual o fato está associado.

Um objeto de conhecimento no SemantiCore pode conter, além do item já descrito e os *default* do *framework*, outros quatro itens, que são: *SensorITEM*, *EffectorITEM*, *RuleITEM* e *ActionPlanITEM*. Cada um desses itens está relacionado a um dos componentes básicos da estrutura do agente no SemantiCore, como será descrito a seguir.

(a)



(b)

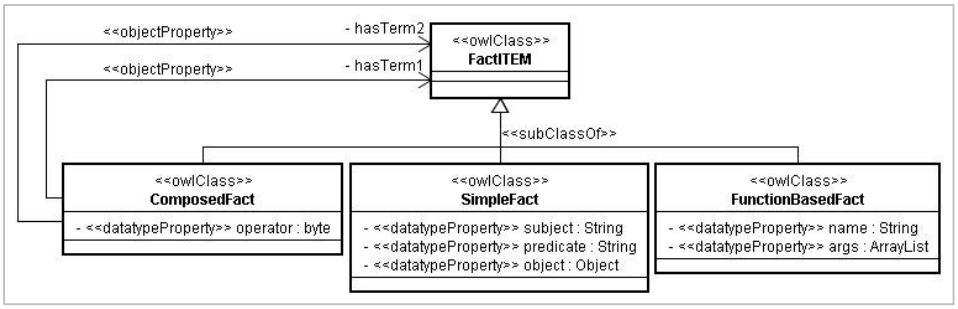


Figura 5.5: A classe *Fact* e suas especializações

```
<FunctionBasedFact rdf:ID="fact1">
  <args rdf:datatype="http://www.w3.org/2001/XMLSchema#string">1_X</args>
  <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">lessThan</name>
  <args rdf:datatype="http://www.w3.org/2001/XMLSchema#string">2_Y</args>
</FunctionBasedFact>
```

Figura 5.6: Representação das informações de um *FuncionBasedFact* (fragmento de código OWL)

O *SensorITEM* tem como elemento um objeto da classe *semanticore.agent.sensorial.Sensor* e está relacionado, portanto, ao componente sensorial do agente. Cada *SensorITEM* indica um tipo de sensor que deve ser instanciado no agente para que ele perceba um determinado conjunto de eventos do ambiente. Todo sensor possui dois tipos de padrão de verificação nas mensagens: o padrão de cabeçalho (*headerPattern*) e o padrão de conteúdo (*contentPattern*). Como os eventos (ou mensagens) disparados no ambiente estão estruturados na forma de ontologias, os padrões do sensor estão associados à classe *Fact* (*semanticore.agent.kernel.information.Fact*). Para a geração da representação ontológica dos sensores são capturados os seguintes aspectos: nome do sensor, nome da classe para instanciação (os sensores são pontos de flexibilidade do SemantiCore) e os padrões de cabeçalho e conteúdo. Quando um objeto de conhecimento é aplicado, o sensor recriado é adicionado à lista de sensores do componente sensorial. De maneira similar, quando o objeto

de conhecimento deve ser removido, todos os sensores contidos em seus itens devem ser removidos da lista de sensores.

O *EffectorITEM* tem uma implementação bastante similar ao *SensorITEM*. Seu elemento, no entanto, remete ao componente efetuador do agente (possui como elemento um objeto da classe *semanticore.agent.effector.Effector*). Cada *EffectorITEM* indica um efetuador que deve ser adicionado ao agente para que possam ser enviados diferentes tipos de mensagens pelo ambiente. Para a representação ontológica deste item, são capturados apenas o nome do efetuador e a classe que o implementa (os efetuadores também são pontos de flexibilidade da arquitetura do SemantiCore).

Embora os efetuadores e sensores sejam elementos que podem ser criados pelos desenvolvedores na arquitetura do SemantiCore, algumas implementações destes itens já são disponibilizadas junto com a plataforma. Como um mesmo sensor ou efetuador pode ser utilizado em diferentes aplicações, acredita-se que suas classes estarão disponíveis na plataforma para instanciação. Isso indica que, quando uma classe não é localizada para a instanciação no agente receptor do conhecimento, o objeto de conhecimento que contém o item é considerado incompatível. Esse problema pode ser resolvido, no entanto, com a implementação de alguma política para que, nesses casos, seja solicitada a classe faltante ao agente que compartilhou o conhecimento.

O item *RuleITEM*, por sua vez, contém um elemento do tipo *Rule* (*semanticore.agent.kernel.information.Rule*). A classe *Rule* representa as regras explicitadas em função do conhecimento de domínio que servem de entrada para o mecanismo decisório do agente. As regras possuem, além de um nome, dois atributos do tipo *Fact* (*semanticore.agent.kernel.information.Fact*): *cause*, que representa fatos ocorridos; e *consequence*, que representa as conseqüências que a ocorrência de cada fato provoca. No momento da aplicação de algum objeto de conhecimento as regras devem ser carregadas na máquina de inferência do componente decisório do agente.

Por último, o *ActionPlanITEM* tem como elemento um objeto da classe *semanticore.agent.execution.model.ActionPlan*. Como já comentado na Seção 4.3.2, cada objeto de conhecimento deve conter apenas um desses itens. No SemantiCore, cada objeto da classe *ActionPlan* possui os seguintes elementos:

- Um identificador (nome) do tipo *string*.
- Um conjunto de ações.

- Um contexto de execução (variáveis e valores) que é compartilhado com todas as ações contidas no plano.

Cada ação contida no plano de ação é um objeto da classe *semanticore.agent.execution.model.Action*, que possui um método abstrato chamado *exec,* onde deve ser inserido o código que o agente irá de fato executar para a resolução de um problema. Em todos os itens do objeto de conhecimento específicos do SemantiCore, as ações são as únicas que possuem um atributo para a inserção do código fonte na ontologia de representação do elemento (atributo *code*). Isso se dá porque as classes com os códigos das ações geralmente não são visíveis a todos os agentes do sistema, visto que freqüentemente são criadas localmente e não são disponibilizadas na plataforma para instanciação. As ações são específicas do domínio do problema e, no caso, do objeto de conhecimento ao qual estão associadas. Como atributos, toda ação possui um nome, uma pré e uma pós-condição (objetos do tipo *Fact*). A existência de pré e pós-condição indicam que é necessária a ocorrência de um fato para disparar uma ação e também que o término de uma ação implica na criação de um fato novo.

Quando um objeto de conhecimento é aplicado, o plano de ação contido em seus itens deve ser relacionado ao objetivo do agente que disparou o processo de captura de conhecimento. Quando é feita essa associação, o plano de ação é adicionado à lista de planos do componente executor do agente. Na remoção do *ActionPlanITEM* o plano deve ser removido tanto do componente executor quanto do objetivo ao qual está relacionado.

Na Figura 5.7 estão representadas as ontologias criadas para os quatro últimos itens descritos. Como se pode observar na figura, para descrever as variáveis adicionadas ao contexto do plano de ação, foi criada uma *owlClass Variable*, que contém duas *datatype properties*: *name* e *value*. Para cada variável presente no contexto do plano, é criado um novo indivíduo da classe *Variable*. Note que a propriedade *value* não possui um tipo de dados definido como alvo. Isso se dá porque as variáveis podem conter diferentes tipos de dados em seus valores.

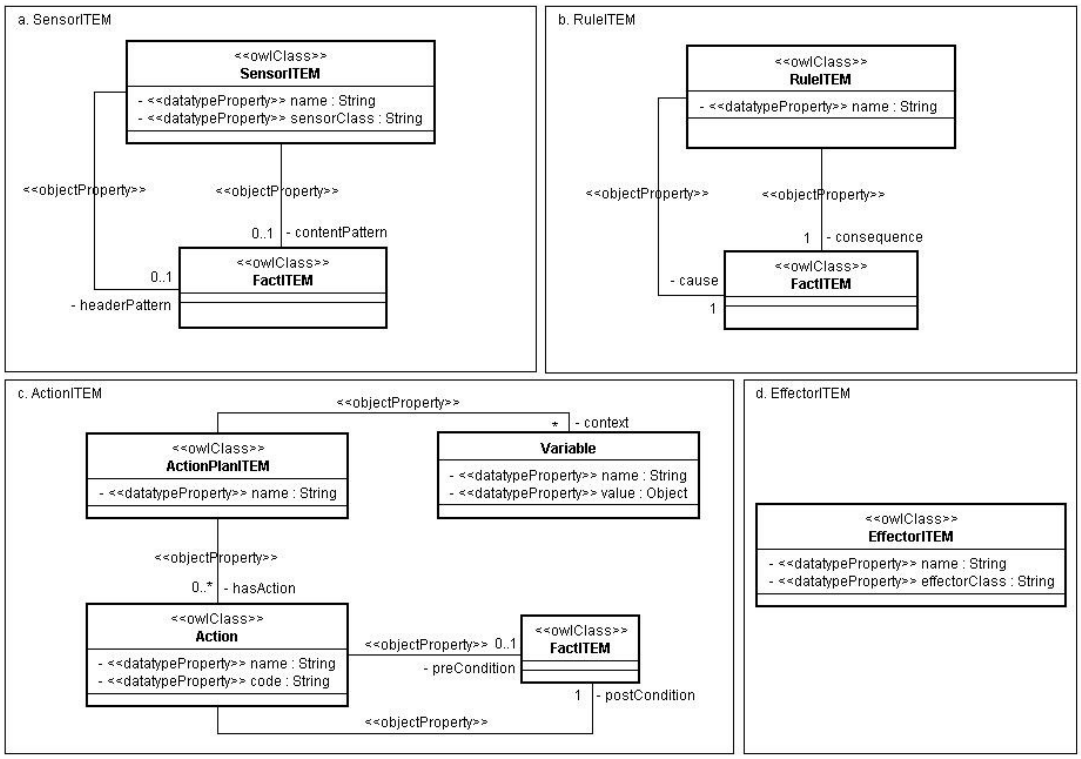


Figura 5.7: Ontologias para os itens *SensorITEM*, *RuleITEM*, *ActionITEM* e *EffectorITEM*

5.2.3 Adaptações requeridas no SemantiCore

Para que o *framework* pudesse ser notificado de alguns eventos ocorridos dentro do agente ou recebidos através do ambiente, foi necessário acrescentar trechos de código em classes específicas do SemantiCore, o que será descrito a seguir.

Conforme o apresentado na Seção 5.2.1 e sintetizado na Tabela 5.1, são quatro os tipos de mensagens recebidas pelo *framework* através do agente ao qual ele está integrado. Para receber as mensagens internas de requisição de conhecimento, foram feitas algumas alteração na classe que representa o objetivo do agente no SemantiCore, a classe *semanticore.domain.model.Goal*. No SemantiCore, cada agente possui uma lista de objetivos e cada objetivo, por sua vez, está relacionado com um plano de ação (classe *semanticore.agent.execution.model.ActionPlan*). Toda vez que um objetivo é iniciado (seja pelo recebimento de fatos do ambiente ou por solicitação do usuário) o seu plano de ação é inserido na fila de planos para execução do componente executor. Porém, considerando-se

que com o *framework* proposto o agente será capaz de procurar conhecimento para atingir determinado objetivo, foi criado mais um construtor na classe *Goal*, que permite que um objetivo seja criado sem ter, necessariamente, um plano de ação associado para a sua execução. Dessa forma, quando é iniciado um objetivo sem plano de ação associado, o *framework* é notificado para localizar o conhecimento necessário, tendo-se como base a ontologia contida no atributo *ontology* de *Goal* (atributo com uma ontologia que traz a descrição semântica do objetivo).

As indicações de término de execução dos planos de ação são sinalizadas ao *framework* pelo componente executor do agente. Toda vez que um plano sai da lista de planos em execução o componente Organizador de Conhecimento deve ser notificado. Nesta notificação, além do identificador do plano, é enviado o seu contexto final de execução.

Já as requisições de conhecimento externas e as respostas a solicitações de conhecimento chegam até o *framework* através do componente sensorial. Para que sejam captadas as mensagens de solicitação e distribuição de conhecimento postadas no ambiente, foi criado o *OrganizerSensor* (subclasse de *semanticore.agent.sensorial.Sensor*). Através deste sensor, as mensagens cujos conteúdos contenham como tipo as *strings* "*requestKnowlede*" ou "*deliverKnowledge*" são encaminhadas ao *framework* para processamento.

Por último, para que o *framework* possa enviar mensagens pelo ambiente, basta criar um objeto do tipo *SemanticMessage* (*semanticore.domain.model.SemanticMessage*) e solicitar ao componente efetuador que transmita essa mensagem (através do método *transmit()*). Neste caso, não é necessário acrescentar qualquer linha de código em classes do SemantiCore.

5.2.4 A base central de conhecimento

Para armazenar o conhecimento homologado do sistema, onde conhecimento homologado significa o conhecimento que já foi verificado e validado por especialistas do domínio, sugere-se a criação de uma base central de conhecimento, representada por um agente da plataforma – o agente Bibliotecário (*Librarian*).

O agente Bibliotecário, assim como os demais agentes do sistema, deverá estar integrado ao *framework* proposto. Seus objetos de conhecimento, no entanto, nunca serão adquiridos por sua própria experiência, visto que ele não deverá ter nenhum objetivo relacionado além do compartilhamento de conhecimento.

O conhecimento recuperado a partir deste agente, por ser homologado, pode ser considerado mais "confiável", tendo preferência no momento da seleção de conhecimento para a aplicação (essa preferência deve ser indicada na política para seleção de objetos de conhecimento para aplicação - *loadPolicy*). Também, pode-se restringir a busca por conhecimento apenas ao Bibliotecário (indicando o seu nome na lista de destinatários – *addresseeList*), o que filtra o conhecimento recebido para determinada solicitação.

Por ser um agente da plataforma, o Bibliotecário deve ser carregado juntamente com os agentes que representam os demais serviços da plataforma (como controle de mensagens e *yellow pages*). Porém, isso não indica que ele deva ter uma estrutura diferenciada dos outros agentes do sistema, mas apenas que ele possui uma função especial. A princípio, não devem ser criadas restrições de acesso para os objetos de conhecimento deste agente, visto que todos devem ser de domínio público.

Para a inclusão e o armazenamento de novos objetos de conhecimento em tempo de execução, sugere-se a criação de interfaces para a criação de objetos de conhecimento no Bibliotecário. Posteriormente, pretende-se definir políticas para a validação do conhecimento adquirido pelos agentes durante suas execuções. Para isso, seria necessário algum tipo de mecanismo no Bibliotecário para a verificação e homologação do conhecimento recebido.

5.3 Considerações sobre o capítulo

Este capítulo destacou aspectos referentes à implementação da arquitetura proposta, descrevendo os padrões e as ferramentas utilizadas no seu desenvolvimento bem como o ambiente de implementação e teste. Foram apresentadas as principais classes do protótipo, com suas características e funcionalidades. Também, foram ressaltados os aspectos

referentes aos itens do objeto de conhecimento e à criação das estruturas ontológicas para as suas descrições.

No próximo capítulo será apresentado o desenvolvimento de uma aplicação, visando demonstrar a viabilidade e os ganhos alcançados com a arquitetura proposta.

6 DESENVOLVENDO APLICAÇÕES COM O *FRAMEWORK* PROPOSTO

O *framework* proposto oferece uma série de facilitadores para organizar o conhecimento de agentes de software, permitindo a recuperação e o compartilhamento de conhecimento nestas entidades. Para ilustrar algumas dessas funcionalidades, este capítulo apresenta o desenvolvimento de uma aplicação que compreende uma solução parcial para o exemplo descrito por Berners-Lee e co-autores em (BERNERS-LEE *et al.*, 2001). A implementação do exemplo foi feita em linguagem Java (J2SDK), no ambiente de desenvolvimento integrado Eclipse[10]. Já para a criação e edição das ontologias utilizou-se o *framework* Jena (versão 2.4).

No exemplo de Berners-Lee e co-autores, agentes interagem entre si de forma a agendar sessões de fisioterapia para um paciente. Assim, o sistema inclui desde a entrada de dados, onde deve ser informado, por exemplo, o nome do paciente, passando pela recuperação do tratamento prescrito e pela análise das clínicas disponíveis.

A implementação dos pontos de flexibilidade do SemantiCore, cuja arquitetura é utilizada como base para a criação dos agentes do sistema, não foi necessária, visto que as implementações disponibilizadas junto com a plataforma se demonstraram suficientes. Assim, foi necessário apenas indicar, na configuração dos agentes, as implementações que seriam utilizadas. Para o mecanismo decisório, utilizou-se o *InferenceEngine* que, com o apoio do Jena, controla o disparo de objetivos e ações do agente, através de regras que definem suas pré-condições (fatos necessários para o início do objetivo ou ação). Para captar as mensagens enviadas através do ambiente, utilizou-se a classe *OWLSensor*, que verifica mensagens com conteúdo em OWL. O *GenericEffector* foi utilizado para permitir o envio de mensagens com diferentes conteúdos através do ambiente. Por último, o *SimpleExecutionEngine* foi utilizado para controlar a execução das ações de um plano.

Este capítulo está organizado da seguinte forma: inicialmente, será apresentada a implementação dos pontos de flexibilidade do *framework* dependentes da aplicação, como os critérios de seleção e as políticas de troca e distribuição de conhecimento. Em seguida, é

[10] http://www.eclipse.org/

descrito o exemplo de Berners-Lee e co-autores e vários aspectos do sistema desenvolvido são apresentados, como a criação dos agentes e dos objetos de conhecimento.

Para a apresentação, foram salientados os aspectos funcionais do problema e a definição das ontologias (tanto dos objetivos dos agentes quanto dos objetos de conhecimento). No decorrer do texto, no entanto, são apresentados alguns trechos de código fonte para auxiliar na ilustração de questões referentes à implementação.

6.1 Implementação dos pontos de flexibilidade dependentes da aplicação

6.1.1 Critérios de distribuição de conhecimento

Para a implementação dos critérios de distribuição de conhecimento foram utilizadas algumas características para a seleção de ontologias apresentadas por Tello e Gómez-Pérez em (TELLO e GÓMEZ-PÉREZ, 2004). Em Tello e Gómez-Pérez (2004) é apresentado o OntoMetric, que é um método para a escolha e comparação de ontologias baseado em uma taxonomia com 160 características chamado *framework* multinível de características.

No *framework* multinível de características, tem-se no primeiro nível da taxonomia cinco fatores: (1) **conteúdo** da ontologia e organização de seus conteúdos; (2) **linguagem** de implementação da ontologia; (3) **metodologia** de desenvolvimento; (4) **ferramenta** utilizada para a construção e edição da ontologia; e (5) **custo** da ontologia no projeto. A princípio, apenas o fator conteúdo será considerado na avaliação dos objetivos pertencentes aos objetos de conhecimento, visto que todos os objetivos são formalizados em OWL, não há uma metodologia de desenvolvimento específica e não há custos associados à obtenção dos objetos de conhecimento, pois mesmo para os objetos restritos que não são de domínio público é prevista a possibilidade de compra.

No fator conteúdo, além das informações relacionadas à ontologia (meta-dados), são citadas várias características utilizadas para medir a adequação de um ontologia

candidata em relação a uma ontologia base. Essas características são agrupadas sob alguns dos principais elementos de uma ontologia, como conceitos, relacionamentos, relações taxonômicas e axiomas. Em se tratando dos conceitos da ontologia, características como presença de conceitos essenciais[11], conceitos essenciais em níveis superiores da taxonomia e o número de conceitos da ontologia são indicados. Já nas características para os relacionamentos contidas na ontologia, há questões como verificação da existência dos relacionamentos (ou propriedades) essenciais e verificação das características destes relacionamentos (como domínio e alvo).

Nas características quanto à taxonomia de conceitos tem-se a verificação da profundidade máxima na hierarquia e também a média de filhos por conceito. Por último, nas características referentes aos axiomas, é verificado, por exemplo, se os axiomas são usáveis para responder as perguntas a partir de deduções e também se eles são úteis para a verificação da consistência da ontologia.

O OntoMetric é um processo manual e subjetivo (dependente do ponto de vista do avaliador) de comparação entre ontologias. Através de uma hierarquia de critérios ponderados, os indivíduos indicam uma pontuação (através de uma escala lingüística) aos nodos folha e essa pontuação vai se propagando até os níveis superiores da hierarquia, onde é atribuída a medida de idoneidade da ontologia candidata as necessidades do projeto. No caso deste trabalho, é atribuída a medida de idoneidade de um objeto de conhecimento as necessidades expressas em uma solicitação de conhecimento.

Para compor os critérios de distribuição de conhecimento foi utilizado apenas um subconjunto das características relacionadas ao fator conteúdo do OntoMetric. Como havia muitas características subjetivas que envolviam, principalmente, o processamento de linguagem natural (para verificar, por exemplo, se os conceitos estavam bem descritos e se essa descrição estava de acordo com os atributos apresentados) a escolha dos critérios foi feita de forma a possibilitar suas avaliações automáticas (sem a interferência humana). Na Tabela 6.1, estão indicadas as quatro características consideradas, sendo duas referentes aos conceitos das ontologias e duas referentes aos seus relacionamentos.

Durante a análise de como automatizar as características apresentadas na Tabela 6.1 foi preciso definir quais os aspectos que seriam verificados nas ontologias e como

[11] Conceitos essenciais são os conceitos que estão na ontologia de entrada para a verificação.

a presença ou não desses aspectos seria medida. De maneira geral, a medida atribuída para cada critério é um valor percentual que indica a taxa de elementos da ontologia base que estão presentes na ontologia candidata. Na verificação dos conceitos essenciais, por exemplo, dado que uma ontologia candidata possui 6 dos 8 conceitos de uma ontologia base, considera-se que ela possui 75% dos conceitos essenciais. Se desses 6 conceitos idênticos apenas 3 estão dispostos no mesmo nível da taxonomia da ontologia base, diz-se que 50% dos conceitos essenciais presentes na ontologia candidata estão no mesmo nível da taxonomia que na ontologia base ou, em outras palavras, a ontologia candidata possui 50% de conceitos essenciais em níveis similares da taxonomia.

Tabela 6.1: Características identificadas para a verificação de similaridade

Características para verificação	
Quanto a conceitos	Presença dos conceitos essenciais
	Conceitos essenciais em níveis similares da taxonomia
Quanto a relacionamentos	Presença dos relacionamentos essenciais
	Conceitos relacionados de forma similar a requerida no sistema (alvo e domínio das propriedades)

Na aplicação, as porcentagens atribuídas aos critérios foram convertidas em uma escala numérica com valores de 1 a 5 (através da criação de uma subclasse de *Converter*), onde cada valor retornado está associado a um grau de adequação em uma escala lingüística (coluna "Grau de adequação" da Tabela 6.2). Uma ontologia candidata que contenha 75% dos conceitos essenciais de uma ontologia de solicitação, por exemplo, possuirá um grau de adequação "alto" no critério "Presença de conceitos essenciais", sendo que o valor retornado será "4".

Tabela 6.2: Conversor crescente de valores percentuais

Porcentagem	**Grau de Adequação**	**Valor de retorno**
[0, 30[	Muito Baixo	1
[30, 50[	Baixo	2
[50, 70[	Médio	3
[70, 90[	Alto	4
[90, 100]	Muito alto	5

Na Figura 6.1 tem-se os critérios definidos dispostos em uma hierarquia. A verificação dos elementos contidos na ontologia candidata e o cálculo da porcentagem são realizados apenas nos nodos folha. Os demais critérios são avaliados de acordo com a soma ponderada de seus subcritérios. Da mesma forma que no OntoMetric, os pesos são atribuídos

pelos usuários. Essa atribuição é feita já no momento da instanciação da classe que representa o critério (no método construtor). Para o desenvolvimento dos exemplos, os pesos foram atribuídos de acordo com o ilustrado na Figura 6.1.Por exemplo, a verificação do critério **C1** ("Quanto a conceitos") é feita através do somatório dos valores atribuídos a C1_1 e C1_2 multiplicados pelos seus pesos (0,8 e 0,2, respectivamente). A pontuação final é dada na avaliação do critério **C** ("Geral"), onde é feito o somatório das avaliações dos subcritérios C1 e C2 multiplicadas pelos seus pesos (neste caso, os dois critérios têm o mesmo peso, que é 0,5).

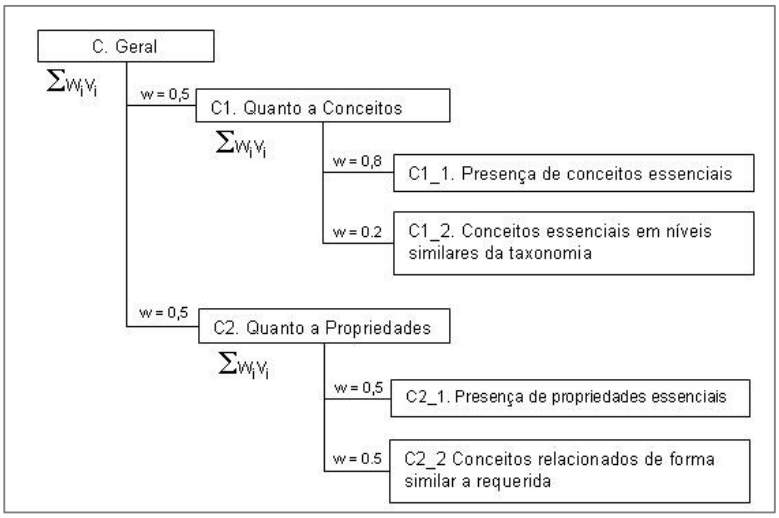


Figura 6.1: Hierarquia de critérios para a distribuição de conhecimento

A seguir, tem-se o código fonte desenvolvido para o critério C1_1 e também um trecho de código com a instanciação e aplicação no *framework* de alguns dos critérios apresentados (ilustrando a instanciação do *hotspot Criterion*). A ligação entre os critérios e o *framework* é realizada através do método *setDeliverCriterion*, que pertence à classe *KnowledgeOrganizerComponent* ("*organizer*", no código, representa um objeto dessa classe).

```
public class Criterion_C1_1 extends Criterion {

    public Criterion_C1_1 (String name, double weight){
        super(name, weight);
    }

    public Object evaluate (Object object)
    {
        int countCandidate = 0; //numero de conceitos da ontologia base presentes
                                //na ontologia candidata
        int countBase = 0; // número total de conceitos da ontologia base

        if (object instanceof ArrayList)
        {
            ArrayList entry = (ArrayList) object;
```

```
            if ((entry.get(0) instanceof OntModel)
                            && (entry.get(1) instanceof OntModel))
            {
                OntModel base = (OntModel) entry.get(0);
                OntModel candidate = (OntModel) entry.get(1);
                for (Iterator i = base.listClasses(); i.hasNext();)
                {
                        OntClass cls = (OntClass) i.next();
                        countBase++;
                        if (candidate.contains(cls, null))
                                countCandidate++;
                }
                return this.converter.measureConverter(new Double((countCandidate
                                *100)/countBase));
            }
        }
        return null;
    }
}
```

```
//Criação dos objetos que representam os critérios
Criterion c = new Criterion_C ("geral", 1);
Criterion c1 = new Criterion_C1 ("conceitos", 0.5);
Criterion c1_1 = new Criterion_C1_1 ("conceitos essenciais", 0.8);
Criterion c2 = new Criterion_C2 ("propriedades", 0.5);

//Associação de um conversor no critério folha da hierarquia
c1_1.setConverter(new NumConverter ());

//Adição de subcritérios
c1.addSubCriterion (c1_1);
c.addSubCriterion(c1);
c.addSubCriterion(c2);

//Atribuição dos critérios de distribuição e aplicação de //conhecimento em um objeto da
classe principal do framework
organizer.setDeliverCriterion ( c );
```

6.1.2 Critérios de aplicação de conhecimento

Além das características utilizadas nos critérios de distribuição (relacionadas a conceitos e relacionamentos das ontologias), nos critérios de aplicação há mais um elemento a ser considerado: o risco relacionado à não aplicabilidade do objeto. Neste trabalho, o risco de não aplicabilidade refere-se, principalmente, à abrangência do conhecimento encapsulado no objeto de conhecimento em relação à necessidade do agente. Por exemplo, tem-se um objeto de conhecimento (OC1) cujo objetivo possui todos os conceitos da ontologia de solicitação (dez conceitos), o que indica um grau muito alto de adequação ao critério C1_1. Porém, na ontologia de OC1, além dos conceitos idênticos aos da ontologia da solicitação, há mais

quinze conceitos, o que indica que muitos outros conhecimentos estão encapsulados no objeto. Quanto maior o número de elementos extras na ontologia do objeto, maior é a probabilidade de o conhecimento não ser aplicável à necessidade do agente, visto que muitos outros elementos não relacionados ao problema serão carregados no agente, dificultando a execução de seu objetivo. Há casos onde o conhecimento adicional não impacta na execução do objetivo, porém, há também casos em que este conhecimento pode impossibilitar a execução, principalmente por falta de dados.

Para exemplificar um caso onde a falta de dados influencia na execução do objetivo, suponhamos a seguinte situação: um determinado agente, AG1, possui um objetivo que envolve a compra de passagens. Na ontologia deste objetivo existem os conceitos "Compra", "Passagem" e "Cidade", como ilustrado na Figura 6.2 (a). Num determinado momento, o agente é notificado da necessidade de compra de uma passagem de Porto Alegre a Curitiba (como indicado na Figura 6.2 (b), que mostra os dados para a execução do objetivo). Depois de solicitar conhecimento para o objetivo descrito, AG1 seleciona um objeto de conhecimento para aplicação cujo objetivo está ilustrado na Figura 6.3. Como pode ser visto na Figura 6.3, o objetivo do objeto de conhecimento selecionado possui uma classe adicional: a "EmpresaTransporte". Se os dados dessa classe forem obrigatórios para a execução de qualquer ação relacionada ao objeto de conhecimento (por exemplo, pode existir uma ação em que dados da empresa de transporte sejam necessários para a verificação da disponibilidade de passagens), não será possível utilizar esse objeto para o objetivo descrito, visto que faltarão dados para a execução.

O cálculo do risco associado ao objeto é feito considerando-se a porcentagem de elementos extras de seu objetivo (em relação à solicitação de conhecimento). A porcentagem calculada indica a taxa de elementos não equivalentes da ontologia candidata de acordo com o seu total de elementos. No exemplo ilustrado anteriormente, cuja ontologia do objeto de conhecimento possuía 25 conceitos e desses conceitos 15 não estavam na ontologia de solicitação, tem-se um percentual de 60% de conceitos extras, o que indica um risco de grau médio, como pode ser visto na Tabela 6.3, que apresenta o risco de aplicabilidade de um objeto de conhecimento através do uso de um conversor para o qual, quanto maior a porcentagem atribuída, menor é o valor de retorno.

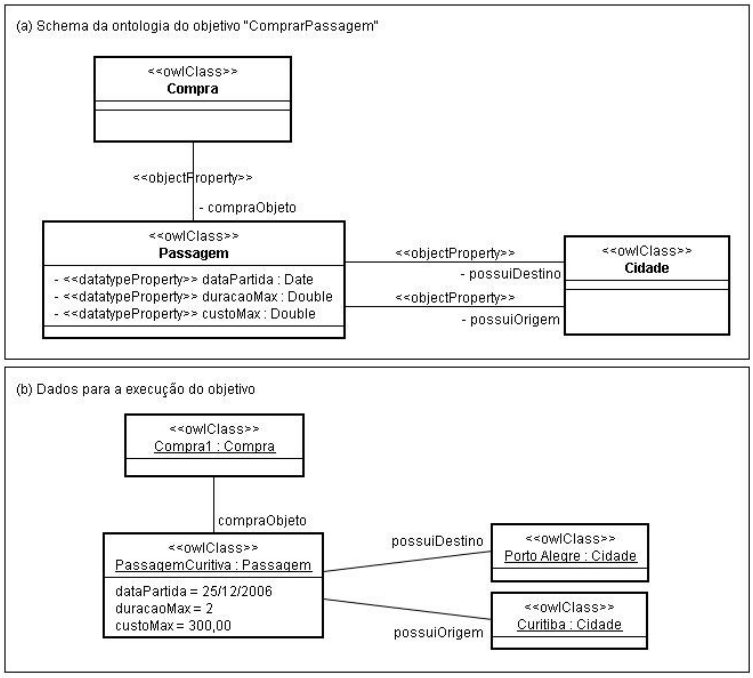


Figura 6.2: Exemplo de *schema* e dos dados para a execução de um objetivo

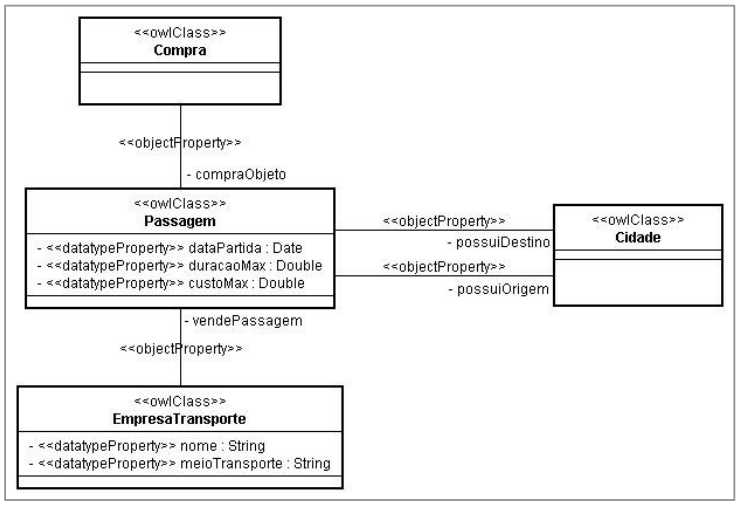


Figura 6.3: Exemplo de objetivo de um objeto de conhecimento

Como pode ser visto na Figura 6.4, que traz a hierarquia de critérios para a aplicação, os critérios para a verificação de conceitos e propriedades extras são subcritérios, respectivamente, dos critérios C1 (“Quanto a conceitos”) e C2 (“Quanto a propriedades”).

Nos critérios de aplicação foram utilizados dois tipos de conversores: (1) conversor crescente (Tabela 6.2) para o qual, quanto maior o valor de entrada (porcentagem calculada nos critérios) maior é o valor final atribuído; (2) conversor decrescente onde, quanto maior o valor de entrada, menor é o valor de retorno. Nos critérios C1_1, C1_2, C2_1 e C2_2 é utilizado o conversor crescente, visto que, quanto mais elementos semelhantes existirem

entre as ontologias, maior deve ser a pontuação final do objeto. Já nos critérios de verificação de risco (C1_3 e C2_3), é utilizado o conversor decrescente onde, quanto maior a incidência de elementos divergentes, menor é o valor de retorno, o que diminui a nota final do objeto.

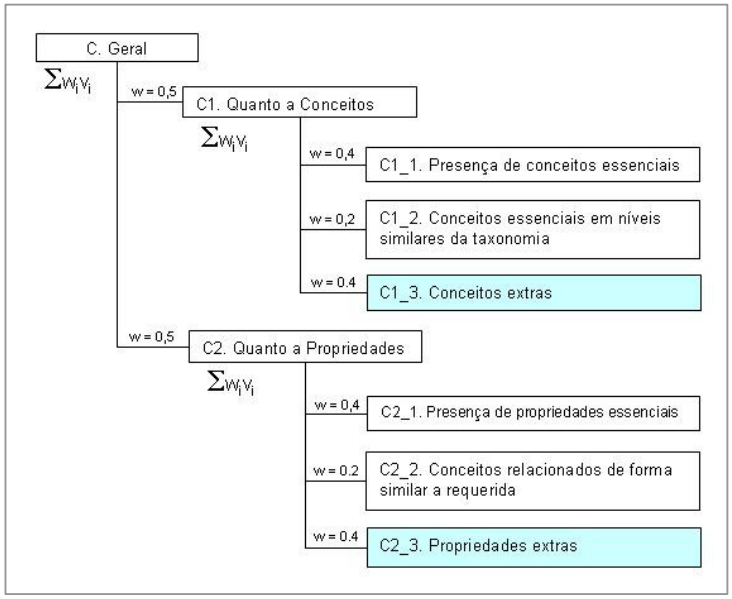


Figura 6.4: Hierarquia de critérios para a aplicação de conhecimento

Tabela 6.3: Risco associado à aplicabilidade de um objeto de conhecimento (conversor decrescente)

Porcentagem de elementos não equivalentes	Risco associado	Valor de retorno
[0, 30[	Muito Baixo	5
[30, 50[	Baixo	4
[50, 70[	Médio	3
[70, 90[	Alto	2
[90, 100]	Muito alto	1

6.1.3 Critério de verificação de execução

Para classificar a execução dos objetos de conhecimento foi criada a *SimpleGoalCriterion*, que é subclasse de *Criterion*. Todos os objetivos dos objetos de conhecimento estão associados a essa classe para a verificação da execução.

No método *evaluate* (herdado de *Criterion*) é verificado o número de sentenças verdadeiras de acordo com o contexto do final da execução do plano. As sentenças são os objetos da classe *Sentence* associados ao objetivo do objeto de conhecimento (classe *KOGoal*). Com o número de sentenças verdadeiras é verificada a porcentagem de sentenças satisfeitas. A porcentagem recuperada, da mesma forma que nos critérios para distribuição e aplicação de conhecimento (apresentados na seção anterior), é transformada para um valor de uma escala numérica com o uso de um conversor crescente (Tabela 6.2) o que indica que, quanto maior a porcentagem atribuída, melhor (mais favorável) é o resultado da execução.

6.1.4 Política de troca

Sendo que a classificação da execução dos objetos de conhecimento é feita em uma escala numérica com valores de 1 a 5, para a avaliação do histórico de execução de um objeto de conhecimento definiu-se a seguinte política de troca: a cada 5 execuções de um mesmo objeto de conhecimento com classificação menor ou igual a 3, deve ser feita a substituição do conhecimento vigente.

Para implementar essa política de troca foi criada uma nova classe, a *MaxOccurPolicy*, que é subclasse de *ChangePolicy*. No seu método *checkHistory*, onde se tem como entrada uma lista com todas as avaliações para um determinado objeto de conhecimento, é feito um somatório das execuções com classificação com valor menor ou igual a 3. Se o resultado do somatório for igual a 5 é retornado o valor verdadeiro, que significa que será iniciado o processo de captura de novos conhecimentos para o objetivo do agente no qual o objeto de conhecimento está relacionado. Caso contrário, é retornado o valor falso, que não dispara nenhuma ação.

6.1.5 Políticas de seleção para distribuição e aplicação de conhecimento

Para a distribuição de conhecimento, considerou-se apenas o envio do objeto mais compatível (com maior pontuação), de acordo com a solicitação recebida, para o agente

solicitante. Porém, há duas exceções: (1) se nenhum dos objetos disponíveis localmente possui uma pontuação maior ou igual a 3 (grau de adequação médio), não serão enviados objetos de conhecimento como resposta à solicitação; e (2) caso haja empate na primeira posição, todos os objetos com maior pontuação são enviados. O agente que representa a base de conhecimento central, no entanto, possui uma política de distribuição um pouco diferente: todos os objetos de conhecimento com pontuação maior que 3 são enviados como resposta ao agente solicitante.

Já na política de aplicação de conhecimento, seleciona-se o objeto de conhecimento com maior pontuação recebido, independentemente do seu valor de adequação. Caso haja incompatibilidade na aplicação de qualquer item do objeto de conhecimento selecionado, remove-se este objeto da lista de objetos recebidos e o método *selectKObjects* (da classe *SelectionPolicy*) é novamente executado. Esse processo é executado até que se possa aplicar de fato um objeto de conhecimento (carregar todos os seus itens) ou que a lista de objetos recebidos esteja vazia. Neste último caso, considera-se que não há conhecimento compatível no sistema para a solicitação efetuada e o objetivo do agente não será executado.

6.2 O exemplo de Berners-Lee e co-autores

Para demonstrar a viabilidade e o uso de algumas características do *framework*, a seguir serão apresentados alguns aspectos da implementação do exemplo apresentado por Berners-Lee e co-autores. Para um melhor entendimento, o texto foi dividido em quatro seções, onde são apresentados, respectivamente, a descrição do problema, os agentes criados, os objetos de conhecimento disponíveis e a solução parcial proposta.

6.2.1 Descrição do problema

Pete e Lucy (dois irmãos) precisam agendar um tratamento médico (sessões de fisioterapia) para a mãe, que será chamada de Marie ao longo do exemplo. Tanto Pete quanto

Lucy possuem agentes pessoais (*personal "Semantic Web agents*") aos quais atribuem diversas tarefas.

A interação para o agendamento das sessões de Marie inicia quando Lucy atribui a seu agente pessoal a tarefa de arranjar automaticamente as sessões de fisioterapia. O agente de Lucy executa, então, várias tarefas: recupera informações do tratamento prescrito com o agente do médico de Marie, procura clínicas capazes de executar o tratamento necessário; seleciona as melhores clínicas considerando a cobertura do plano de saúde de Marie (se a clínica está cadastrada junto ao plano de saúde), o posicionamento da clínica em um *ranking* (só deverão ser selecionadas clínicas com pontuação "excelente" ou "muito bom") e a distância da casa de Marie até a clínica (a clínica deve estar em um raio de 20 milhas); por último, é apresentado um plano com os horários das consultas e as informações de transporte. Feito isso, o agente de Lucy entra em contato com o agente de Pete para expor o plano proposto e verificar se ele é satisfatório.

Pete, no entanto, rejeita o plano devido à localização da clínica indicada e o horário das sessões (para levar a mãe às sessões, Pete necessitaria dirigir na hora do *rush*) e solicita a seu agente que desenvolva um plano diferente que se ajuste melhor a sua agenda, considerando a localização e o horário. Depois de instantes, o agente de Pete apresenta um novo plano com uma clínica com melhor localização e horários – mas há duas advertências: (1) Pete deve reagendar um compromisso marcado como "baixa importância" (o que não é um problema para Pete); e (2) a clínica selecionada não consta na lista de "Fisioterapia" do plano de saúde (mas o agente indica que o tipo de serviço e a ligação da clínica com o plano de saúde foram seguramente verificados por outros meios). Lucy aceita o novo plano sugerido.

6.2.2 Criação dos agentes do sistema

A solução proposta para o exemplo descrito conta com nove diferentes agentes no SemantiCore. O **AgLucy** é o agente pessoal de Lucy e possui uma série de objetivos, sendo que um deles é denominado "SelecionarClinicas", através do qual o agente pode encontrar uma clínica apropriada para a realização de um tratamento prescrito por um médico a um paciente. Para a execução deste objetivo, é necessário informar ao agente o nome do

paciente, o nome do médico onde deve ser recuperado o tratamento, o nome do plano de saúde (caso seja necessário restringir a busca por clínicas associadas a um plano de saúde), as coordenadas do endereço do paciente e a distância máxima da casa do paciente à clínica selecionada (caso haja essa restrição). A Figura 6.5 ilustra, em notação UML, a ontologia deste objetivo do agente.

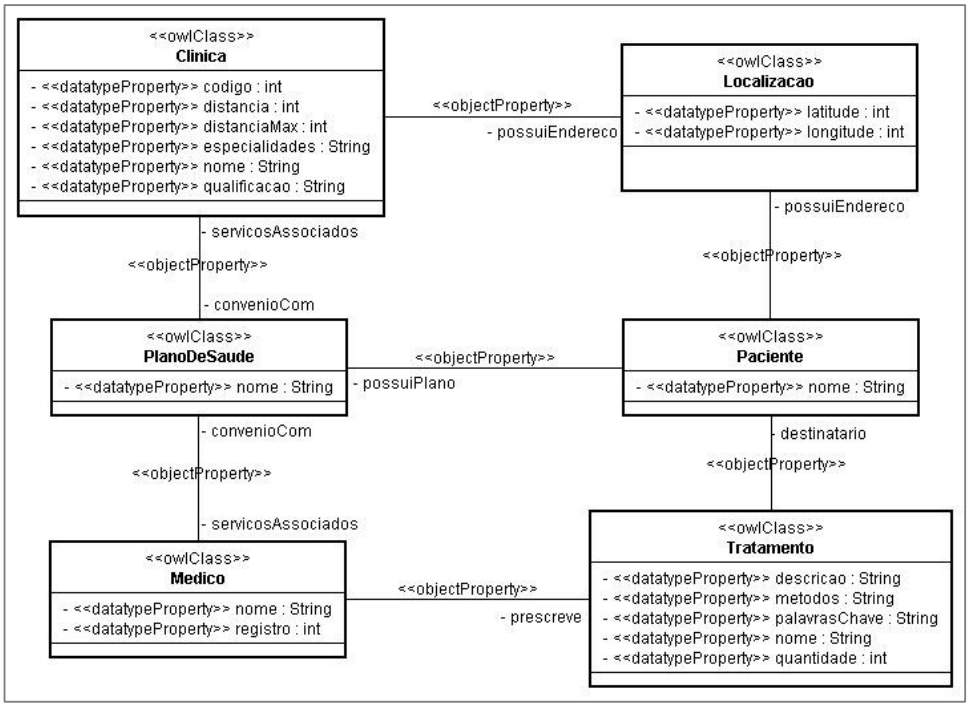


Figura 6.5: Objetivo "SelecionarClinicas"

O **AgPete**, por sua vez, representa o agente pessoal de Pete. Ele também possui, entre os seus objetivos, o objetivo "SelecionarClinicas". O **AgDrLee**, é o agente que representa o médico de Marie. Para prover informações sobre os tratamentos prescritos, AgDrLee possui um objetivo denominado "InformarTratamento", que é disparado cada vez que chega uma solicitação de informações de tratamento de um determinado paciente. Os dados dos tratamentos dos pacientes são armazenados localmente através de instâncias da estrutura ontológica que representa o tratamento prescrito (ilustrada na Figura 6.6). Cada instância desta ontologia representa as informações do tratamento de um determinado paciente. O objetivo "InformarTratamento" é disparado com a chegada dos seguintes fatos:

```
(?M ns:tipo "RecuperarTratamento")
(?X rdf:type ns:Paciente)
(?X ns: nome ?nome)
```

Como pode ser visto nos fatos, “InformarTratamento” só é disparado quando for recebida uma mensagem onde o conteúdo contenha um indivíduo “?M” cujo tipo (*datatype property* ns:tipo) possua como valor a *string* “RecuperarTratamento” e um indivíduo da classe Paciente (“ns:Paciente”) com um nome associado (variável “?nome”). O “ns” de “ns:tipo” refere-se a uma indicação do *namespace*[12] da ontologia através de um prefixo. O *namespace* define um vocabulário controlado que identifica um conjunto de conceitos de forma única.

Quando um objetivo do agente inicia sua execução, seu plano de ação é automaticamente carregado no componente executor. O objetivo “RecuperarTratamento” está associado a um plano de ação que contém apenas uma ação, a “EnviarTratamento”. Nesta ação, é verificado o tratamento existente para o paciente cujo nome disparou a execução do objetivo e é enviada uma resposta com todas as informações do tratamento ao agente solicitante.

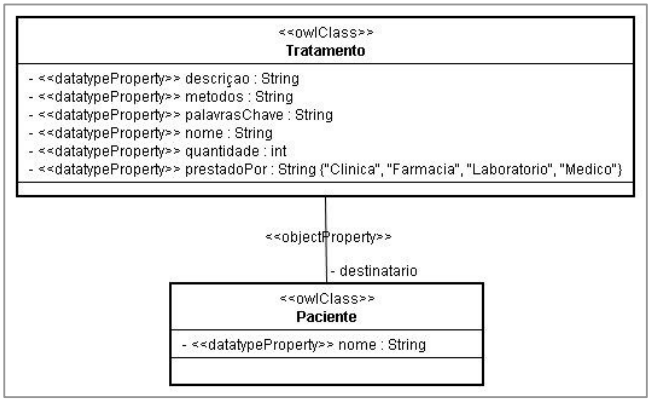


Figura 6.6: Estrutura ontológica para a prescrição de tratamentos médicos

A seguir está o código inserido no método *setup* de AgDrLee (subclasse de *semanticore.domain.model.SemanticAgent* do SemantiCore) para a criação dos fatos que compõem a pré-condição, do plano de ação e das ações do objetivo “RecuperarTratamento”.

```
//Criação de um plano de ação para associar ao objetivo
ActionPlan plan = new ActionPlan ( "plan" );

//Criação da ação cujo código permite o envio das informações do //tratamento de
determinado paciente
AcaoEnviarTratamento a = new AcaoEnviarTratamento ("EnviarTratamento", null);
```

[12] Neste trabalho, todas as ontologias foram definidas utilizando a URL http://semanticore.pucrs.br como *namespace*.

```
plan.addAction(a);

//Criação dos fatos que compõem a pré-condição do objetivo
SimpleFact sf1 = new SimpleFact("?M", ns + "tipo", “RecuperarTratamento”);
SimpleFact sf2 = new SimpleFact("?X", "rdf:type", ns + "Paciente");
SimpleFact sf3 = new SimpleFact("?X", ns + "nome", "?nome");
ComposedFact cf1 = new ComposedFact (sf1, sf2);
ComposedFact cf2 = new ComposedFact (cf1, sf3);

//Criação do objetivo RecuperarTratamento
//Construtor da classe Goal (owner: SemanticAgent, onto: OntModel, //
             plan: ActionPlan, preCondition: Fact)
Goal recuperarTratamento = new Goal (this, tratamentoSchema, plan, cf2);

//adição do objetivo criado ao agente
this.addGoal (recuperarTratamento);
```

AgPlanoSaude representa o agente do plano de saúde de Marie no sistema e possui uma lista de especialidades médicas (como fisioterapia, cirurgia geral, obstetrícia, entre outros) onde, para cada especialidade há uma série de clínicas e médicos associados. Com a chegada de uma solicitação de clínicas para uma determinada especialidade (contidas nas palavras-chave do tratamento), é acionado o objetivo “BuscarClinicas” que, ao ser executado (execução do plano de ação associado), recupera todas as clínicas disponíveis na especialidade solicitada e as envia ao agente solicitante. Para ser disparado o objetivo “BuscarClinicas”, o seguinte conjunto de fatos deve ser recebido por AgPlanoSaude:

```
(?M ns:tipo “RecuperarClinicas”)
(?T rdf:type ns:Tratamento)
(?T ns:palavrasChave ?pc)
```

Em AgPlanoSaude, as informações de cada clínica estão explicitadas dentro de uma estrutura ontológica, que deve ser uma instância da ontologia representada na Figura 6.7. No momento do envio das informações das clínicas, toda a ontologia de dados da clínica selecionada (ou clínicas selecionadas) é enviada.

O **AgAvaliador** possui um *ranking* com a classificação de clínicas e serviços. Cada vez que é recebida uma lista com um ou mais prestadores de serviços, AgAvaliador verifica a classificação dos prestadores de serviços recebidos e as envia para o agente solicitante. Para tanto, ele possui um objetivo denominado “VerificarClassificação” que inicia sempre que é recebida uma mensagem cujo conteúdo contenha um indivíduo com uma propriedade “tipo” com o valor “VerificarClassificacao”.

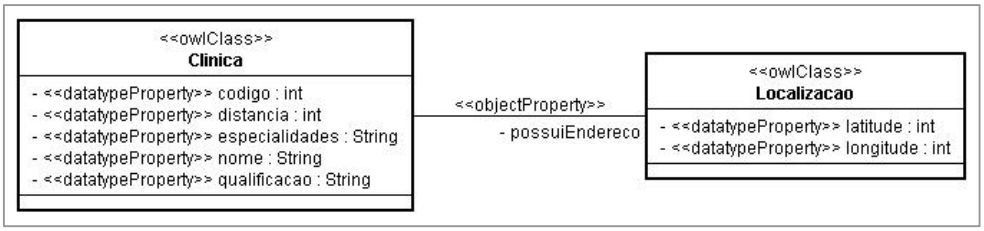


Figura 6.7: Estrutura ontológica para as informações das clínicas

AgAnne é um agente adicional (que não está especificado na descrição do problema) que foi criado para exemplificar o mecanismo de busca de conhecimentos do *framework*. Ele representa a secretária de um médico e possui um bom conhecimento de como agendar consultas e tratamentos. Ainda para exemplificar as funcionalidades do *framework*, foram criados quatro agentes que representam quatro diferentes clínicas, cada qual com suas especialidades. Esses agentes são chamados de **AgClinica1**, **AgClinica2**, **AgClinica3** e **AgClinica4**. Todo agente que representa uma clínica possui um objetivo denominado "OferecerServicos", que é inicializado sempre que é recebida a solicitação de um serviço de algum agente do sistema. Caso a clínica ofereça o serviço solicitado, é enviada uma notificação ao agente solicitante.

Por fim, foi criado o agente ***Librarian*** que, como especificado na Seção 5.2.4, representa a base de conhecimento central do *framework*. Esse agente não possui nenhum objetivo associado, visto que irá apenas compartilhar objetos de conhecimento.

6.2.3 Criação dos objetos de conhecimento

Foram criados, ao todo, quatro objetos de conhecimento no sistema. Três desses objetos ("KO-1", "KO-2" e "KO-3") estão disponíveis no agente *Librarian* e são de domínio público (não possuem restrições associadas). O quarto objeto de conhecimento ("KO-4") pertence ao agente AgAnne.

O objeto de conhecimento "KO-1" encapsula o conhecimento necessário para a compra e venda de passagens. Já o objeto "KO-2" encapsula o conhecimento para o agendamento de tratamentos e consultas, sem a verificação do plano de saúde do paciente ou sua localização. O objeto "KO-3", também envolve o agendamento de tratamentos, dessa vez,

considerando o plano de saúde do paciente e sua localização. Por último, o objeto "KO-4" encapsula um conhecimento diferenciado para o agendamento de tratamentos, onde as solicitações são feitas diretamente aos agentes que representam as clínicas. Como cada clínica, ao responder uma solicitação de serviço, indica os planos de saúde aceitos, existem regras para a verificação da cobertura do plano de saúde em "KO-4". Os objetivos dos quatro objetos de conhecimento são mostrados nas figuras 6.8, 6.9, 6.10 e 6.11 (a descrição completa de todos os itens dos objetos de conhecimento criados é feita no Apêndice A).

O objetivo de "KO-1" possui as seguintes sentenças para a verificação da execução (objetos da classe *Sentence*):

Sentence	***Subject***	***Operator***	***Value***
S1:	"duracaoF"	"<"	"duracaoMax"
S2:	"custoF"	"<"	"custoMax"
S3:	"dpF"	"="	"dataPartida"
S4:	"mtF"	"="	"meioTransporte"
S5:	"custoF"	"<"	"custoMax"/2

Nas sentenças de "KO-1" tanto os sujeitos (*subject*) quanto os valores (*value*) são variáveis que devem ser recuperadas no contexto de execução do plano finalizado. As variáveis presentes no campo *value* armazenam as informações ou fatos responsáveis pela inicialização do objetivo e devem estar acessíveis, portanto, para a verificação de sua execução.

Os objetivos de "KO-2", "KO-3" e "KO-4" possuem apenas uma sentença de verificação associada. De acordo com esta sentença (escrita abaixo), o objetivo é satisfeito se o número de clínicas sugeridas para o usuário for maior que zero, ou seja, se pelo menos uma clínica for encontrada.

Sentence	***Subject***	***Operator***	***Value***
S1:	"numClinicas"	">"	"0"

No código escrito a seguir, tem-se um trecho da implementação de AgAnne e da criação dos itens de seu objeto de conhecimento, o "KO-4".

```
public class AgAnne extends SemanticAgent
{
    public AgAnne(Environment env, String agentName)
    {
        super(env, agentName);
    }

    protected void setup()
    {

      //Indicação das implementações para os hotspots do SemantiCore
      setDecisionEngine ( new InferenceEngine ( getDecisionComponent()));
```

```
        setExecutionEngine ( new SimpleExecutionEngine (getExecutionComponent()));
        setEffectorEngine ( new GenericEffector ( getEffectorComponent ( ) ) );
        addSensor ( new OWLSensor ( "sensorOrganizerComponent", null, null ) );


        //==== Setup do componente organizador ====
        KnowledgeOrganizerComponent organizer = new KnowledgeOrganizerComponent ( this,
                                                "organizer" );
        getSensorialComponent ( ).addTransmissionComponent ( "organizer" );
        addComponent ( "organizer", organizer );


        //Indicação das políticas de distribuição e aplicação
        organizer.setDeliverPolicy ( new BiggerScorePolicy ( ) );
        organizer.setLoadPolicy ( new SimpleLoadPolicy ( ) );


        //...Criação e indicação dos critérios para distribuição e carregamento

        organizer.start();
        // ==== Fim setup do componente organizador


        //Criação do elemento KOGoal
        OntModel selecionarClinicas = ModelFactory.createOntologyModel (
                                    OntModelSpec.OWL_MEM, null );
        selecionarClinicas.read ( "file:schemas/SelecionarClinicas.owl", null );

        KOGoal goal = new KOGoal (selecionarClinicas);
        goal.setCriterion (new SimpleGoalCriterion ("SimpleGoalCriterion", 1.0));
        goal.addSentence (new Sentence ("numClinicas", ">", "0"));


        //Criação de um sensor
        Sensor sensor = new OWLSensor ("sensorKO-4", new SimpleFact ("?id",
                                "http://semanticore.pucrs.br#to", agentID), null);

        //... Criação dos outros elementos dos itens do objeto de conhecimento

        //Criação de um objeto de conhecimento
        KnowledgeObject ko4 = new KnowledgeObject ("KO-4");
        ko4.addItem(new GoalITEM (goal));
        ko4.addItem(new SensorITEM (sensor));

        //... adição de outros itens

        //Adição do objeto de conhecimento KO-4 a base de conhecimento do agente
        organizer.knowledgeBase.storeKnowledge (ko4);
    }
}
```

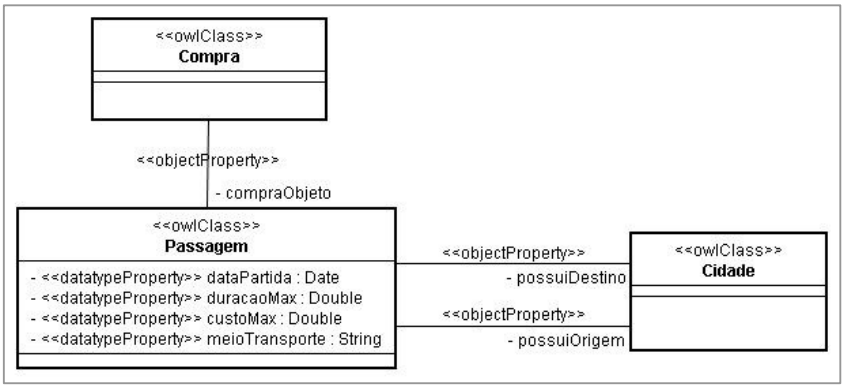


Figura 6.8: Estrutura ontológica do objetivo de "KO-1"

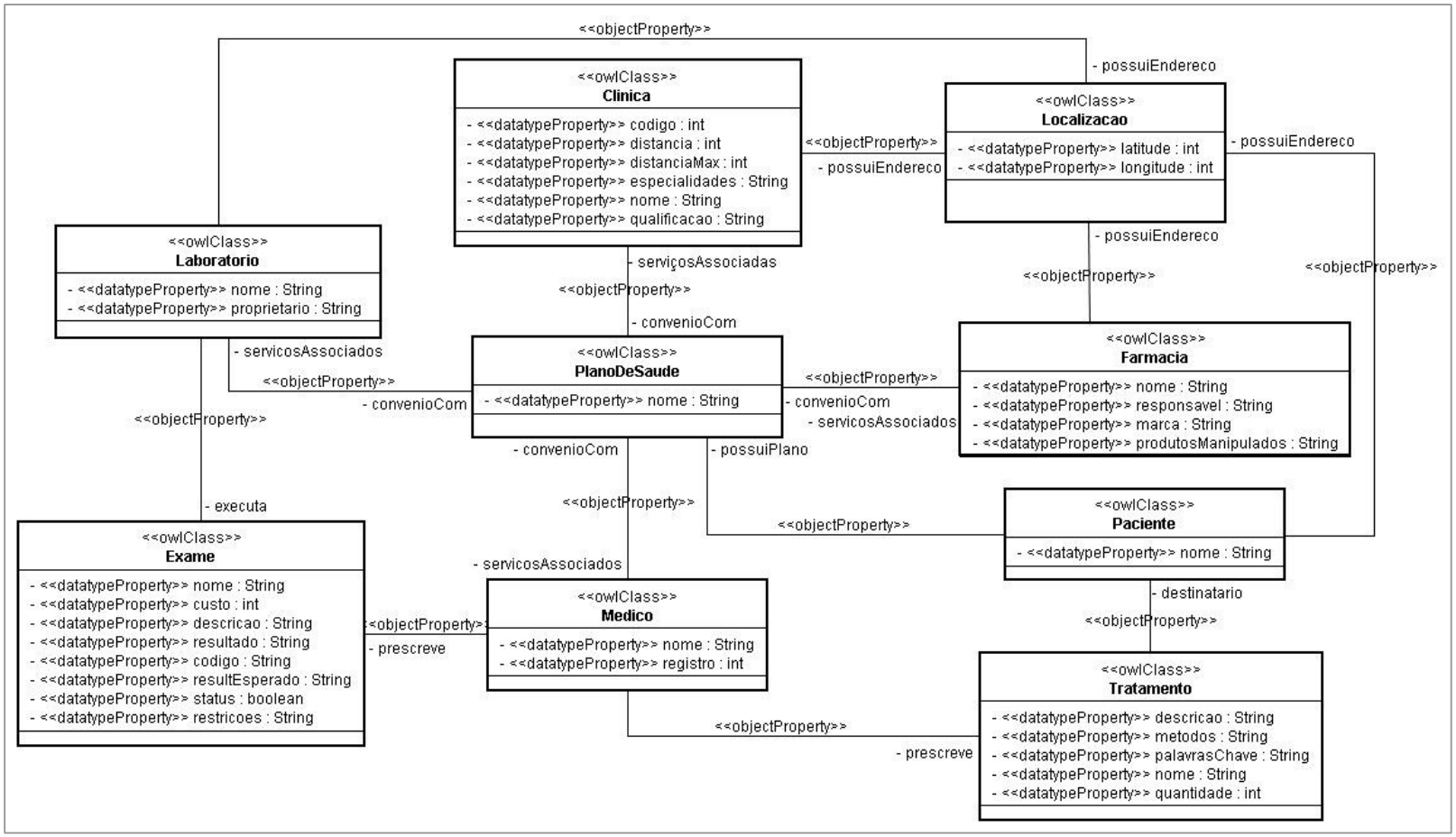


Figura 6.9: Estrutura ontológica do objetivo de “KO-3”

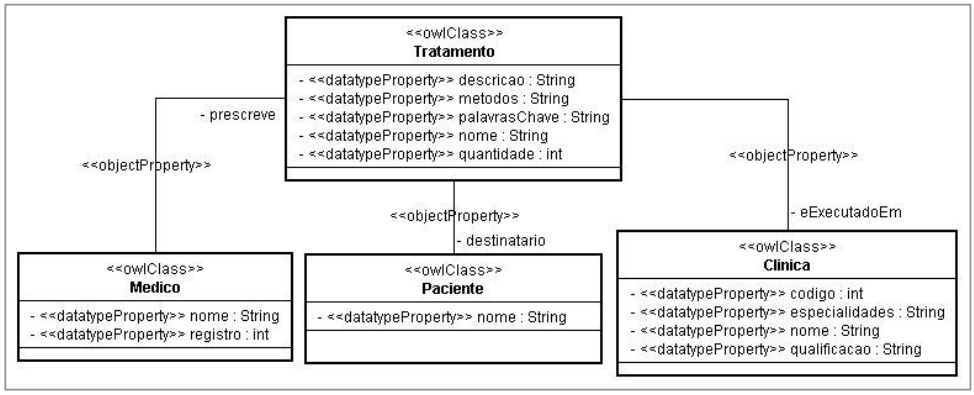


Figura 6.10: Estrutura ontológica do objetivo de "KO-2"

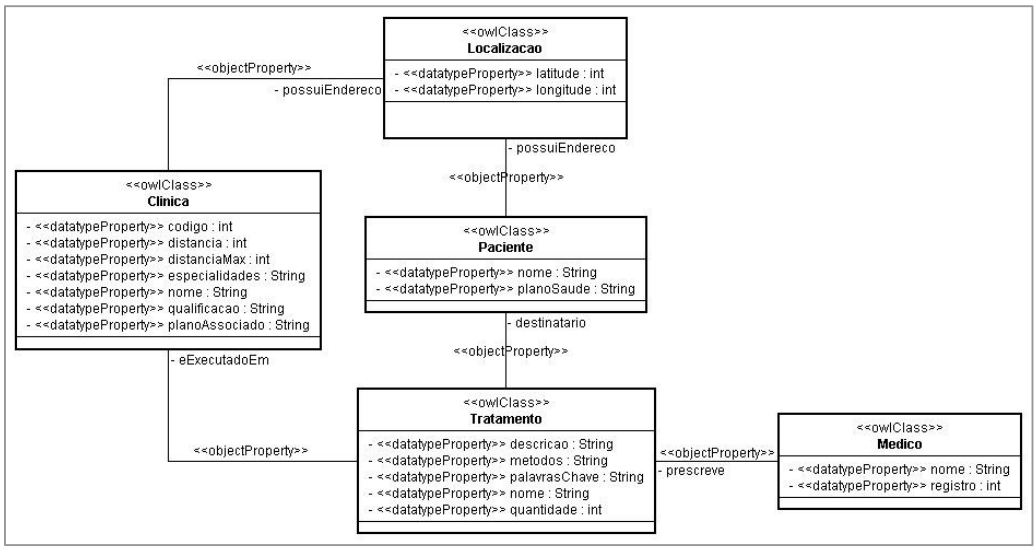


Figura 6.11: Estrutura ontológica do objetivo de "KO-4"

6.2.4 Descrição da solução parcial proposta[13]

A seguir, será descrita uma proposta inicial de implementação para o exemplo apresentado por Berners-Lee e co-autores, considerando-se, principalmente, aspectos referentes à busca, compartilhamento e aplicação de objetos de conhecimento. O uso de diferentes objetos de conhecimento permite que AgPete, por exemplo, seja melhor sucedido

[13] O desenvolvimento desta aplicação visa apenas exemplificar as funcionalidades do *framework* desenvolvido e não propor uma solução completa para o exemplo apresentado por Berners-Lee. Neste contexto, não foram verificados, nesta primeira implementação, aspectos referentes à disponibilidade de horários nas agendas de Lucy e Pete, ou seja, apenas são propostas as melhores clínicas para o tratamento de Marie, mas os horários não são agendados automaticamente.

ao encontrar uma clínica que não está relacionada com a especialidade “Fisioterapia” do plano de saúde de sua mãe.

Como já mencionado, a execução inicia quando Lucy solicita a AgLucy o início da execução do objetivo “SelecionarClinicas”, tendo-se como base os seguintes fatos (dados de entrada para a execução do objetivo):

```
1.  (ns:Paciente1 rdf:type ns:Paciente)
2.  (ns:Paciente1 ns:nome “Marie”)
3.  (ns:Medico1 rdf:type ns:Medico)
4.  (ns:Medico1 ns:nome “DrLee”)
5.  (ns:PlanoMarie rdf:type ns:PlanoSaude)
6.  (ns:PlanoMarie ns:nome “PlanoSaude”)
7.  (ns:Paciente1 ns:possuiPlano ns:PlanoMarie)
8.  (ns:LocMarie rdf:type ns:Localizacao)
9.  (ns:LocMarie ns:latitude “20”)
10. (ns:LocMarie ns:longitude “40”)
11. (ns:Paciente1 ns:possuiEndereco ns:LocMarie)
12. (ns:ClinicaX rdf:type ns:Clinica)
13. (ns:ClinicaX ns:distanciaMax “40”)
```

Note que, inicialmente, é criado um indivíduo da classe “ns:Paciente” cuja propriedade nome (“ns:nome”) tem como valor a *string* “Marie” – linhas 1 e 2. De forma similar, nas linhas 3 a 6 são criados indivíduos da classe “ns:Medico” e “ns:PlanoSaude”, com seus respectivos nomes. Nas demais linhas são indicadas a localização de Marie e a distância máxima permitida entre a localização de Marie e a localização da clínica selecionada.

Para demonstrar o mecanismo de busca de conhecimento do *framework*, definiu-se que AgLucy não possui, *a priori*, o conhecimento necessário para atingir o objetivo “SelecionarClinicas”, ou seja, o objetivo não está relacionado a nenhum plano de ação. Também, AgLucy possui uma política de solicitação de conhecimento restrita, onde o conhecimento faltante é procurado apenas na base local de conhecimento e no agente *Librarian* (na lista de destinatários de mensagens de solicitação – *addresseeList* - consta apenas “*local*” e “*Librarian*”).

Depois de fazer a solicitação de conhecimento para o objetivo “SelecionarClinicas”, AgLucy recebe do agente *Librarian* dois objetos de conhecimento: “KO-2” e “KO-3”. Ambos os objetos receberam pontuação maior que 3 na verificação dos critérios de distribuição de *Librarian* e, devido a sua política de distribuição (envio de todos os objetos com pontuação maior que 3), foram enviados como resposta a AgLucy.

Na Tabela 6.4 está ilustrado o número total de conceitos e propriedades da solicitação enviada e também dos objetos de conhecimento disponíveis em *Librarian* (dados

utilizados na computação da porcentagem). Já a Tabela 6.5 ilustra a pontuação atribuída a cada critério para os objetos de conhecimento "KO-2" e "KO-3". As pontuações de "KO-1" não estão representadas na tabela, porque o conhecimento encapsulado neste objeto não tem qualquer relação com o conhecimento solicitado (não há nem conceitos nem propriedades semelhantes). Neste caso, todos os critérios são avaliados com um grau de adequação "muito baixo" e a nota final atribuída ao objeto é "1".

Tabela 6.4: Número total de conceitos e propriedades da solicitação e dos objetos de conhecimento

	Total de conceitos	**Total de propriedades**
Solicitação ("SelecionarClinica")	6	19
KO-1	3	7
KO-2	4	12
KO-3	9	26

Tabela 6.5: Pontuação atribuída aos objetos pelos critérios de distribuição em *Librarian*

	KO-2		**KO-3**	
Critérios	**%**	**Grau**	**%**	**Grau**
C1_1	66,67	3	100	5
C1_2	100	5	100	5
C2_1	57,89	3	100	5
C2_2	90,0	5	84,0	4
C1	-	3,4	-	5
C2	-	4	-	4,5
C	**-**	**3,7**	**-**	**4,75**

Como pode ser visto na Tabela 6.6, "KO-3", mesmo possuindo um número considerável de elementos extras (o que diminui sua pontuação final devido ao risco de não aplicabilidade), ainda possui um grau de adequação maior que "KO-2" quando pontuado através dos critérios de aplicação de AgLucy. "KO-3" foi, portanto, o objeto selecionado para aplicação.

Tabela 6.6: Pontuação atribuída aos objetos pelos critérios de aplicação em AgLucy

	KO-2		**KO-3**	
Critérios	**%**	**Grau**	**%**	**Grau**
C1_1	66,67	3	100	5
C1_2	100	5	100	5
C1_3	0	5	33,0	4
C2_1	57,89	3	100	5
C2_2	90,0	5	84,0	4
C2_3	8,33	5	38,0	4
C1	-	4,2	-	4,6
C2	-	4,2	-	4,4
C	**-**	**4,2**	**-**	**4,5**

Depois de aplicar “KO-3”, AgLucy executou as seguintes ações encadeadas (contidas no plano de ação associado a “KO-3”):

- **Adquirir prescrição médica**: é através desta ação que o agente do usuário entra em contato com o agente do médico para recuperar o tratamento prescrito. No exemplo, AgLucy entra em contato com AgDrLee para recuperar o tratamento de Marie.
- **Recuperar lista de serviços do plano de saúde**: com base nas informações adquiridas a partir do tratamento prescrito, é solicitado ao agente que representa o plano de saúde do paciente (AgPlanoSaude) uma lista com as serviços disponíveis (que atendam as especialidades informadas no tratamento). A indicação de busca por serviços e não especificamente por clínicas se dá porque “KO-3” não trabalha apenas com a recuperação de informações de clínicas, mas informações de farmácias, médicos e clínicas.
- **Recuperar qualificação**: é através desta atividade que as informações dos serviços disponíveis no plano de saúde são encaminhadas ao agente AgAvaliador para serem avaliadas. AgAvaliador retorna uma lista com os prestadores de serviços juntamente com as suas qualificações (excelente, muito bom, bom, regular, ruim, muito ruim).
- **Calcular distância**: com base nas informações de localização do prestador de serviço recuperado e na endereço do paciente (já disponível no agente) é calculada a distância da casa do paciente à clínica.
- **Selecionar clínicas candidatas**: de posse da lista de prestadores de serviços, de suas qualificações e da distância até casa de paciente, são selecionados os melhores serviços para o tratamento. De acordo com a descrição do exemplo, para um serviço (no caso uma clínica) ser selecionado, ele deve estar localizado a uma distância menor que 20 milhas da casa do paciente e sua qualificação deve ser “excelente” ou “muito bom”. Como os serviços são recuperados diretamente do plano de saúde, não é necessário refazer a verificação da cobertura do plano. Depois de selecionadas as clínicas, o usuário é notificado para que faça a aceitação do plano.

Lucy, sem ter nenhuma objeção às clínicas selecionadas (foram selecionadas as clínicas "Clinica1" e "Clinica3" representadas, respectivamente por AgClinica1 e AgClinica3), solicita que AgLucy envie a lista a AgPete para que Pete tome conhecimento do plano. Como foram encontradas clínicas para a execução do tratamento, a sentença para a classificação da execução do objetivo "SelecionarClinicas" (apresentada na Seção 6.2.3) foi satisfeita, o que resulta em uma classificação de execução com valor "5" (100% das sentenças foram satisfeitas).

Conforme a descrição do problema, Pete não fica satisfeito com a lista recuperada por AgLucy e solicita que AgPete refaça a busca. AgPete, da mesma forma que AgLucy, possui o objetivo "SelecionarClinicas" sem nenhum conhecimento associado. AgPete, no entanto, possui uma política de busca de conhecimento diferente de AgLucy: suas solicitações de conhecimento são enviadas para todos os agentes registrados na plataforma. Assim, como resposta a sua solicitação, AgPete recebe do agente *Librarian* os objetos de conhecimento "KO-2" e "KO-3", de AgLucy o objeto de conhecimento "KO-3" e de AgAnne, o objeto "KO-4".

Como o conhecimento solicitado por AgPete é idêntico ao solicitado por AgLucy, os objetos de conhecimento disponíveis em *Librarian* terão a mesma pontuação apresentada na Tabela 6.5 para a solicitação de AgPete. A pontuação atribuída a "KO-3", em AgLucy, também é idêntica à pontuação dada por *Librarian*, visto que ambos possuem os mesmos critérios para distribuição de conhecimento. Já em AgAnne, o objeto de conhecimento "KO-4" (que possui 5 conceitos e 18 propriedades) é avaliado conforme o indicado na Tabela 6.7.

Tabela 6.7: Pontuação atribuída à "KO-4" pelos critérios de distribuição em AgAnne

	KO-4	
Critérios	**%**	**Grau**
C1_1	83,33	4
C1_2	100	5
C2_1	84,0	4
C2_2	93,0	5
C1	-	4,2
C2	-	4,5
C	**-**	**4,35**

Ao avaliar os quatro objetos de conhecimento recebidos, AgPete seleciona o objeto "KO-4", que alcança a maior pontuação com o uso dos critérios de aplicação, como pode ser visto na Tabela 6.8. Observe que "KO-4", mesmo tendo uma pontuação menor que

"KO-3" na distribuição (avaliado com os critérios de distribuição), possui um menor risco de não aplicabilidade, o que acaba aumentando o sua pontuação na avaliação para a aplicação.

Tabela 6.8: Pontuação atribuída aos objetos pelos critérios de aplicação em AgPete

	KO-2		KO-3		KO-4	
Critérios	**%**	**Grau**	**%**	**Grau**	**%**	**Grau**
C1_1	66,67	3	100	5	83,33	4
C1_2	100	5	100	5	100	5
C1_3	0	5	33,0	4	0	5
C2_1	57,89	3	100	5	84,0	4
C2_2	90,0	5	84,0	4	93,0	5
C2_3	8,33	5	38,0	4	15,0	5
C1	-	4,2	-	4,6		4,6
C2	-	4,2	-	4,4		4,6
C	**-**	**4,2**	**-**	**4,5**		**4,6**

Com o uso de "KO-4", a recuperação da lista de clínicas não é feita com base na cobertura do plano de saúde (como era em "KO-3", que foi utilizado por AgLucy), mas sim através de solicitações de serviços a todos os agentes do sistema, como pode ser visto na descrição das ações de seu plano de ação associado:

- **Adquirir prescrição médica**: da mesma forma que em "KO-3", é através desta ação que o agente do usuário entra em contato com o agente do médico para recuperar o tratamento prescrito para o paciente.
- **Solicitar serviços**: utilizando as palavras-chave para clínicas/especialistas presentes no tratamento prescrito, é enviada uma mensagem a todos os agentes do sistema questionando quem está apto a executar o serviço requerido. Os agentes aptos respondem a mensagem através de uma ontologia que contém o nome, os tipos de atividades executadas, os planos de saúde aceitos e a localização da clínica.
- **Recuperar qualificação**: da mesma forma que "KO-3", é através desta atividade que as informações das clínicas recuperadas são encaminhadas ao agente AgAvaliador para serem avaliadas.
- **Calcular distância**: também, de forma idêntica que em "KO-3", através desta ação é calculada a distância entre a clínica e a casa do paciente.
- **Selecionar clínicas:** de posse da lista de clínicas, de suas qualificações e da distância da casa do paciente, são selecionadas as melhores clínicas

> para o tratamento. Para uma clínica ser selecionada, conforme a descrição do problema, ela deve estar a uma distância menor que 20 milhas da casa do paciente, sua qualificação deve ser "excelente" ou "muito bom" e deve ter cobertura do plano de saúde do paciente. Depois de selecionadas as clínicas, é solicitada a aceitação do plano.

AgPete, utilizando o objeto de conhecimento "KO-4" identificou uma clínica a mais do que AgLucy (foram selecionadas as clínicas "Clinica1", "Clinica2" e "Clinica3" representadas, respectivamente, por AgClinica1, AgClinica2 e AgClinica3), isto porque AgPlanoSaude, quando consultado por AgLucy, verificou apenas as clínicas presentes na especialidade "Fisioterapia" e a "Clinica2", erroneamente, encontra-se apenas na lista de "Centros médicos", mesmo dispondo de serviços de fisioterapia.

De acordo com a descrição do problema, Pete fica satisfeito com a nova lista de clínicas encontradas pelo seu agente e aceita a indicação de "Clinica2" para o tratamento de sua mãe. Selecionada a clínica, ainda é necessário agendar as sessões de fisioterapia, o que poderia ser feito automaticamente com o uso de outros objetos de conhecimento.

6.3 Considerações sobre o capítulo

O exemplo de aplicação descrito neste capítulo procurou mostrar, em linhas gerais, as primitivas do *framework* proposto e a forma de usá-las na definição dos objetos de conhecimento e na criação dos agentes. Como pôde ser visto ao longo do capítulo, o uso do *framework*, uma vez implementados os pontos de flexibilidade, não traz um aumento significativo de trabalho aos desenvolvedores, salvo as questões referentes à divisão do conhecimento em objetos de conhecimento e à definição de uma ontologia para representar o conhecimento encapsulado, aspectos esses que poderiam ser considerados complexos não do ponto de vista de desenvolvimento, mas de entendimento do problema.

Com o uso do *framework* é possível indicar uma série de objetivos aos agentes durante suas inicializações sem necessariamente definir como que esses objetivos serão executados (como no objetivo "SelecionarClinicas"). No momento que um determinado objetivo for de fato executado, o conhecimento necessário é então localizado e carregado

automaticamente. Como o conhecimento recuperado muitas vezes tende a ser apenas similar ao conhecimento solicitado, pode haver incompatibilidade entre o conhecimento recuperado e o objetivo do agente, o que piora a classificação da execução. Porém, se isso ocorrer, o conhecimento carregado será substituído por outro conhecimento, através do uso do histórico de execução.

O uso de objetos de conhecimento pode permitir, de fato, um melhor aproveitamento do conhecimento existente, visto que um mesmo objeto pode ser utilizado por vários agentes e em várias situações. Também, a existência de pontos de flexibilidade para a avaliação e seleção dos objetos de conhecimento possibilita que eles sejam avaliados e selecionados de acordo com os mais diferentes fatores.

Durante a descrição do exemplo, não foram salientadas as questões referentes ao histórico de execução e a política de troca, devido a não aplicabilidade destes conceitos no exemplo utilizado. Posteriormente, pretende-se desenvolver uma aplicação mais abrangente, com mais agentes interagindo e negociando a fim de atingir seus objetivos.

7 CONCLUSÕES E TRABALHOS FUTUROS

Considerando-se a importância da GC para prover um ambiente em que exista compartilhamento de conhecimento com suporte ao armazenamento e recuperação de informações, o presente trabalho apresentou o desenvolvimento de um *framework* para a organização do conhecimento de agentes de software que possibilita, entre outras coisas, o reuso do conhecimento existente, visto que o conhecimento pode ser compartilhado entre os agentes e utilizado em diferentes situações e contextos, e a substituição do conhecimento ineficiente, através da análise do histórico de execução.

Para a apresentação, o trabalho foi estruturado dentro de três fases distintas. Na primeira fase, foram esclarecidos aspectos das tecnologias relacionadas ao trabalho e também foi feito um levantamento na literatura sobre abordagens relacionadas à organização do conhecimento de agentes de software, a fim de identificar trabalhos relacionados. A segunda fase foi caracterizada pela descrição de aspectos, tanto conceituais quanto de implementação, do *framework* proposto. Na terceira e última fase, o desenvolvimento de uma aplicação foi apresentado.

Para iniciar a apresentação dos aspectos conceituais do *framework*, foi descrito o processo de GC utilizado como guia, esclarecendo as características envolvidas com cada uma de suas atividades. A capacidade de adquirir novos conhecimentos, por exemplo, sugere que os agentes podem identificar uma necessidade de conhecimento, representar essa necessidade de forma que seja compreendida pelos outros agentes do sistema, e selecionar e aplicar o conhecimento mais similar recebido. Já para que haja distribuição de conhecimento, os agentes devem ser capazes de receber uma solicitação de conhecimento, verificar qual o conhecimento disponível mais similar à solicitação recebida e enviar o conhecimento selecionado ao agente solicitante.

A estruturação do conhecimento disponível nos agentes foi feita através dos *objetos de conhecimento*. Para a representação tanto dos objetos de conhecimento quanto das mensagens para compartilhamento de conhecimento, foram utilizadas ontologias. De acordo com Staab e co-autores, “ontologias constituem o meio para juntar todos os subprocessos de conhecimento (criação, captura, recuperação, acesso, e uso de conhecimento)” (STAAB *et al.*, 2001).

Para a definição de alguns aspectos do *framework*, trabalhou-se com a idéia de que os agentes possuem objetivos e que esses objetivos, por sua vez, devem estar relacionados a um conjunto de ações ou comportamentos que permitam que o agente execute tarefas a fim de atingir o objetivo especificado. No momento da aplicação de um objeto de conhecimento, considerou-se que o conhecimento contido no objeto cobria totalmente o objetivo do agente ao qual ele estava sendo relacionado, ou seja, executando-se as ações contidas no objeto de conhecimento poder-se-ia atingir o objetivo especificado. Dessa forma, um objetivo do agente está relacionado a apenas um objeto de conhecimento por vez. No entanto, outras abordagens poderiam ser implementadas imaginando-se, por exemplo, que para o alcance de um objetivo pode ser necessário o conhecimento de diferentes objetos de conhecimento, sendo que cada objeto contribuiria parcialmente para a execução.

Algumas vezes, mesmo com o objetivo do objeto de conhecimento altamente similar ao objetivo do agente, pode haver incompatibilidade entre eles. Isso ocorre, principalmente, quando o conhecimento presente no objeto de conhecimento é mais abrangente do que o especificado no objetivo do agente. Nestes casos, alguns itens do objeto de conhecimento podem ter sua execução bloqueada ou paralisada por falta de dados. Adaptar os itens do objeto de conhecimento ao objetivo do agente está fora do escopo deste trabalho.

O modelo conceitual proposto, juntamente com a implementação das primitivas do *framework*, constitui a principal contribuição deste trabalho, auxiliando na formalização e no compartilhamento de conhecimento em agentes de software. Em uma visão mais detalhada, as contribuições deste trabalho são:

- Identificação de aspectos da implantação de um processo de GC em uma arquitetura de agentes.
- Definição de um modelo conceitual para o uso de um processo de GC em arquiteturas de agentes.
- Implementação das primitivas do *framework*, o que possibilita sua integração a plataformas de desenvolvimento de SMAs e o desenvolvimento de aplicações para validar o modelo proposto.
- Apresentação do desenvolvimento de aplicações, o que visa guiar os projetistas na definição e implementação dos pontos de flexibilidade do *framework*.

A não implementação do fluxo de criação e representação de novos conhecimentos (Fluxo 1 do processo de GC utilizado), se deu principalmente pelos desafios envolvidos com a criação e validação automatizada de novos conhecimentos. Avaliar se um conhecimento é mais útil que os já existentes não é uma tarefa trivial, visto que os resultados da aplicação ou do uso de determinado conhecimento estão fortemente relacionados com o contexto no qual o agente ou indivíduo está inserido no momento da execução da tarefa. Assim, trabalhos futuros centram-se em estudar, principalmente, questões relacionadas com a aquisição e organização de novos conhecimentos em agentes, considerando-se seus contextos de execução e também suas experiências passadas (histórico de execução).

Outros trabalhos futuros seriam a definição e implementação de um estudo de caso mais abrangente, a exploração do mapeamento do *framework* sobre outras plataformas para o desenvolvimento de SMAs e a definição de um método para orientar os desenvolvedores na criação de agentes e de objetos de conhecimento de acordo com as primitivas do *framework* proposto.

REFERÊNCIAS BIBLIOGRÁFICAS

AAMODT, A.; PLAZA, E. (1994). "Case-based reasoning: foundational issues, methodological variations and system approaches". AI Communications, vol. 7-1, March 1994, pp. 39–59.

AL-SAKRAN, H. (2006). "An Agent-based Architecture for Developing E-learning System". Information Technology Journal, vol. 5-1, 2006, pp. 121-127.

ANTONIOU, G.; HARMELEN, F. (2004). "A Semantic Web Primer". Cambridge: MIT Press, 2004, 272 p.

BELLIFEMINE, F.; CAIRE, G.; TRUCCO, T.; RIMASSA, G. (2005). "JADE Programmer's Guide". Capturado em: http://sharon.cselt.it/projects/jade/doc/programmersguide.pdf, Novembro 2005.

BERNERS-LEE, T.; HENDLER, J.; LASSILA, O. (2001). "The Semantic Web". Scientific American Magazine, May 2001, 10p.

BLOIS, M.; LUCENA, C. (2004). "Multi-Agent Systems and the Semantic Web - The SemanticCore Agent-based Abstraction Layer" In: Proceedings of Sixth International Conference on Enterprise Information Systems (ICEIS), Porto, 2004, pp.263 – 270.

CRANEFIELD, S. (2001). "UML and the Semantic Web". In: Proceedings of the International Semantic Web Working Symposium (SWWS), Palo Alto, 2001. Capturado em: http://www.semanticweb.org/SWWS/program/full/paper1.pdf, Novembro 2005.

DACONTA, M.; OBRST, L.; SMITH, K. (2003). "The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management". Indiana: John Wiley & Sons, 2003, 328p.

DAVENPORT, T.; PRUSAK, L. (1998). "Conhecimento empresarial: como as organizações gerenciam o seu capital intelectual". Rio de Janeiro: Campus, 1998, 237p.

DAVIES, J.; DUKE, A.; STONKUS, A. (2002). "OntoShare: Using Ontologies for Knowledge Sharing". In: Proceedings of the WWW Workshop on Semantic Web, Hawaii, 2002, 26p.

DESPRES, C.; CHAUVEL, D. (1999). “Mastering Information Management: Part Six – Knowledge Management”. Financial Times, vol. 4-6, March 1999, pp. 2-3.

DEVEDZIC, V. (2002). “Understanding ontological engineering”. Communications of the ACM, vol. 45-4, April 2002, pp. 136-144.

DOAN, B-L.; BOURDA, Y.; BENNACER, N. (2004). "Using OWL to Describe Pedagogical Resources". In: 4th IEEE International Conference on Advanced Learning Technologies (ICALT 2004), Finland, 2004, pp. 916-917.

ESCOBAR, M.; LEMKE, A. P.; BLOIS, M. (2006). “SemantiCore 2006: Permitindo o Desenvolvimento de Aplicações baseadas em Agentes na Web Semântica”. In: Workshop on Software Engineering for Agent-Oriented Systems (SEAS), 2006, Florianópolis, pp. 72-82.

FERBER, J. (1999). “Multi-agent systems: an introduction to distributed artificial Intelligence”. Boston: Addison-Wesley, 1999, 528p.

FERBER, J. (2005). “MadKit User's Guide”. Capturado em: http://www.madkit.org/madkit/doc/userguide/userguide.html, Novembro 2005.

FERBER, J.; GUTKECHT, O.; MICHEL, F. (2005). “MadKit Development Guide”. Capturado em: http://www.madkit.org/madkit/doc/devguide/devguide.html, Novembro 2005.

FERNANDES, A.; MOURA, A.; PORTO, F. (2003). “An Ontology-based approach for organizing, sharing and querying learning-objects on the web”. In: 14th International Workshop on Database and Expert Systems Applications, Praga, 2003. IEEE Computer Society 2003, vol. 2736. pp. 604-609.

FREITAS, F.; STUCKENSCHMIDT, H.; NOY, N. (2005). “Ontology Issues and Applications: Guest Editorial”. Journal of the Brazilian Computer Society, vol. 11-2, Novembro 2005, pp. 5-16.

GARTNER GROUP (1998). "Knowledge Management Scenario: Trends and Directions for 1998-2003", Tech. Rep., Stamford, March 1998.

GENESERETH, M. R.; SINGH, N.; SYED, M. (1995). “A distributed and anonymous knowledge sharing approach to software interoperation”. International Journal of Cooperative Information Systems, vol. 4-4, 1995, pp. 339-367.

GEROIMENKO, V. (2003). “Dictionary of XML technologies and the Semantic Web”. London: Springer-Verlag, 2003, 250 p.

GRUBER, T. (1993). “Towards Principles for the Design of Ontologies Used for Knowledge Sharing”. International Journal of Human and Computer Studies, vol. 43-5/6, 1993, pp. 907-928.

GRUBER, T. (1996). “What is an Ontology?”. Capturado em: http://wwwksl. stanford.edu/kst/what-is-an-ontology.html, Setembro 2005.

GUO, L.; ROBERTSON, D.; CHEN-BURGER, Y. (2005). "A Generic Multi-agent System Platform For Business Workflows Using Web Services Composition". In: Proceedings of 2005 IEEE Intelligent Agent Technology, France, 2005, pp. 301-307.

GUTKNECHT, O.; FERBER, J. (2000). “Madkit: A generic multi-agent platform”. In: 4th International Conference on Autonomous Agents, Barcelona, 2000, pp. 78-79.

HENDLER, J. (2001). “Agents and the Semantic Web”. IEEE Intelligent Systems, Mar./Apr. 2001, pp. 30-37.

HOUARI, N.; FAR, B. (2004). “Application of Intelligent Agent Technology for Knowledge Management Integration”. In: Proceedings of IEEE ICCI, Victoria, 2004, pp. 240-249.

JADE - Java Agent DEvelopment Framework (2006). Capturado em: http://jade.tilab.com/, Maio 2006.

JENA - A Semantic Web Framework for Java (2006). Capturado em: http://jena.sourceforge.net/, Agosto 2006.

JESS - Java Expert System Shell (2006). Capturado em: http://herzberg.ca.sandia.gov/jess/, Agosto 2006.

KUCZA, T. (2001) “Knowledge Management Process Model”. Capturado em: http://www.inf.vtt.fi/pdf/publications/2001/p455.pdf, Maio 2005.

LAWTON, G. (2001). "Knowledge Management: Ready for Prime Time?". IEEE Computer, vol. 34-2, Feb. 2001, pp. 12-14.

LEE, D.; LEE, G. (2003). “Knowledge Management for Computational Problem Solving”. Journal of Universal Computer Science, vol. 9-6, Dec. 2003, pp.563-570.

LEMKE, A. P. (2005). “Apoio Automatizado a Processos de Gestão de Conhecimento”. Trabalho Individual I, Mestrado em Ciência da Computação, PUCRS, 2005, 51p.

LI, W. (2002). “Intelligent Information Agent with Ontology on the Semantic Web”. In: Proceedings of the 4th World Congress on Intelligent Control and Automation, Shanghai, 2002, pp. 1501 - 1504.

LO, C.; NG, K.; LU, Q. (2002). "CJK Knowledge Management in Multi-agent m-Learning System", In: Proceeding of The First International Conference on Machine Learning and Cybernetics (ICMLC'2002), Beijing, China, November 2002, pp. 1983-1986.

MadKit - a Multi-Agent Development Kit (2005). Capturado em: http://www.madkit.org/, Novembro 2005.

MAEDCHE, A. (2002). “Ontology Learning for the Semantic Web”. Massachusetts: Academic Publishers, 2002, 272p.

MATHI, K. (2004). “Key Success Factors for knowledge management”, Dissertação de Mestrado, University of Applied Sciences/ Fh Kempten, Alemanha, 2004.

NISSEN, M. (1999). "Knowledge-based knowledge management in the reengineering domain". Decision Support Systems, vol. 27-1, Nov. 1999, pp. 47 - 65.

NOY, N.; MCGUINNESS, D. (2001), “Ontology Development 101: A Guide to Creating Your First Ontology”. Technical Report KSL-01-05 and SMI-2001-0880, Stanford Knowledge Systems Laboratory and Stanford Medical Informatics, March 2001.

OBITKO, M.; MARIK, V. (2002), “Ontologies for Multi-Agent-Systems in Manufacturing Domain”. In: Proceedings of the 13th International Workshop on Database and Expert Systems Applications, France, Sept. 2002, pp. 597- 602.

O'LEARY, D. (1998) “Using AI in Knowledge Management: Knowledge Bases and Ontologies”. IEEE Intelligent Systems, vol. 13- 3, May/Jun. 1998, pp. 34-39.

OMG - Object Management Group (2000). “Agent Technology – the green paper – version 1.0”. Capturado em: http://www.jamesodell.com/ec2000-08-01.pdf, Agosto 2005.

OpenCybele (2005). Capturado em: http://www.opencybele.org/, Novembro 2005.

PLAZA, E. (2005). “Cooperative Reuse for Compositional Cases in Multi-agent Systems”. In: 6th International Conference on Case-Based Reasoning, ICCBR 2005, Chicago, USA, August 2005, Lecture Notes in Computer Science, vol. 3620, pp 382-396.

PLAZA, E.; ONTAÑÓN, S. (2003). “Cooperative Multiagent Learning”. In: Proceedings of Adaptative Agents and Multi-Agent Systems, 2003, pp. 1-17.

POWERS, S. (2003). “Practical RDF”. Cambridge: O'Reilly, 2003, 350p.

PROTÉGÉ - Ontology Editor and Knowledge Acquisition System (2005). Capturado em: http://protege.stanford.edu/, Julho 2005.

RIBEIRO, M. B. (2002). “Web Life: Uma arquitetura para a implementação de sistemas multi-agentes para a Web”. Tese de Doutorado, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 2002, 204 p.

RIESBECK, C.; SCHANK, R. (1989). “Inside Case-Based Reasoning”. New Jersey: Lawrence Erlbaum Associates, 1989, 448 p.

RYBINSKI, H.; RYZKO, D. (2003). "Knowledge sharing in default reasoning based multi-agent systems". In: IEEE/WIC International Conference on Intelligent Agent Technology (IAT'03), Halifax, Canadá, 2003, pp. 576-579.

SEMANTIC WEB (1998). Capturado em: http://www.w3c.org/ 2001/sw/, Março 2005.

STAAB, S.; STUDER, R.; SCHNURR, H.; SURE, Y. (2001). “Knowledge processes and ontologies”. IEEE Intelligent Systems, vol. 16-1, Jan./Feb. 2001, pp. 26-34.

STADER, J.; MACINTOSH, A. (1999). “Capability modelling and knowledge management”. In: Applications and Innovations in Intelligent Systems VII, 1999, Springer-Verlag, pp 33-50.

TELLO, A.; GÓMEZ-PÉREZ, A. (2004), “ONTOMETRIC: A Method to Choose the Appropriate Ontology”. Journal of Database Management, vol. 15-2, 2004, pp 1-18.

TODA, Y.; YAMASHITA, M.; SAWAMURA, H. (2001). “An argument-based agent system with KQML as an agent communication language”. In: Proceedings of Pacific Rim International Workshop on Multi-Agents (PRIMA 2001), Taipei, Taiwan, 2001, pp. 48-62.

VALENTE, G. (2004). “Artificial Intelligence Methods in Operational Knowledge Management”. Tese de Doutorado, Departamento de Informática, Universidade degli Studi di Torino, 2004, 139 p.

W3C (2006a). “RDF – Resource Description Framework”. Capturado em: http://www.w3.org/RDF/, Junho de 2006.

W3C (2006b). “OWL - Web Ontology Language”. Capturado em: http://www.w3.org/2004/OWL/, Junho de 2006.

WEBER, R.; WU, D. (2004). “Knowledge Management for Computational Intelligence Systems”. In: Proceedings of Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE 2004), Tampa, USA, 2004, pp. 116-125.

WEBER, R.; KAPLAN, R. (2003). “Knowledge-Based Knowledge Management”. International Series on Advanced Intelligence, vol. 4-1, July 2003, pp 151-172.

WEISS, G. (1999). “Multiagent systems: a modern approach to distributed artificial intelligence”. Cambridge: The MIT Press, 1999, 619p.

WEISS, G. (1995). “Adaptation and Learning in Multi-Agent Systems: Some Remarks and a Bibliography”. In: Proceedings of IJCAI’95 Workshop, Montreal, Canada. Alemanha: Springer-Verlag, 1995, pp. 1-21.

WOOLDRIDGE, M.; JENNINGS, N.; KINNY, D. (1999). “A methodology for agent oriented analysis and design”. In: Proceedings of International Conference on Autonomous Agents, Seattle, EUA, 1999, pp. 69-76.

WOOLDRIDGE, M. (2002). “An Introduction to Multiagent Systems”. Chichester: John Wiley and Sons Ltda, 2002, 366 p.

ZACK, M. (1999). “Managing codified knowledge”. Sloan Management Review, vol. 40-4, Summer 1999, pp. 45-58.

ZHU, C.; WANG, Z.; LIN, D.; DING, P.; SHENG, H. (2004). "Ontology Mapping For Interaction in Agent Society". In: Proceedings of the IEEE International Conference on Services Computing (SCC'04), Shanghai, 2004, pp. 531- 535.

ZHU, Q.; HICKS, J.; PETROV, P.; STOYEN, A. (2003). “The Topologies of Knowledge Intensive Multi-Agents Cooperation”. In: Proceedings of the IEEE Knowledge Intensive Multi-Agent Systems Conference, Boston, 2003, pp.741-746.

APÊNDICE A

- Itens dos objetos de conhecimento criados -

Tabela A.1: Itens do objeto de conhecimento "KO-1"

<table>
<tr><th>Instância/Item</th><th colspan="2">Atributos da instância</th></tr>
<tr><td>descriptorKO-1/
DescriptorITEM</td><td colspan="2">- kOName: "KO-1"
- kOType: "domínio"
- description: "Esse objeto de conhecimento tem como objetivo a compra de uma passagem de uma determinada origem para um determinado destino, em determinado dia, com custo e duração máximos".
- status: "acabado"
- domain: "Compra de passagens"
- author: "Ana Paula Lemke"
- creationDate: "10/12/2006"</td></tr>
<tr><td>goalKO-1/
GoalITEM</td><td colspan="2">- hasOntology: vide Figura 6.8.
- hasVerificationSentence:
("duracaoF" "<" "duracaoMax")
("custoF" "<" "custoMax")
("dpF" "=" "dataPartida")
("mtF" "=" "meioTransporte")
("custoF" "<" "custoMax"/2
- criterion:
criterionClass: "organizer.hostpots.SimpleGoalCriterion"
name: "SimpleGoalCriterion"
weight: "1"</td></tr>
<tr><td>sensorKO-1/
SensorITEM</td><td colspan="2">- name: "sensorKO-1"
- sensorClass: "semanticore.agent.sensorial.hotspots.OWLSensor"
- headerPattern:
(?id http://semanticore.pucrs.br#to agentID) e
(?tipoMensagem rdf:type http://semanticore.pucrs.br#Proposal)</td></tr>
<tr><td rowspan="3">planKO-1/
ActionPlanITEM</td><td colspan="2">- name: "ComprarPassagem"
- context:
name: "duracaoF"
name: "duracaoMax"
name: "custoF"
name: "custoMax"
name: "dpF"
name: "dataPartida"
name: "mtF"
name: "meioTransporte"</td></tr>
<tr><td rowspan="2">- hasAction</td><td>name: "EnviarSolicitacao"
postCondition:
(?request http://semanticore.pucrs.br#status "aguardandoProp")</td></tr>
<tr><td>name: "ConfirmarCompra"
preCondition:
((?request http://semanticore.pucrs.br#status "aguardandoProp") e
(?request http://semanticore.pucrs.br#custoMax ?custoMax) e
(?proposal http://semanticore.pucrs.br#custo ?custo) e
lessThan (?custo, ?custoMax)) ou
((?request http://semanticore.pucrs.br#status "aguardandoProp") e
(?request http://semanticore.pucrs.br#custoMax ?custoMax) e
(?request http://semanticore.pucrs.br#duracaoMax ?duracaoMax) e
(?proposal http://semanticore.pucrs.br#custo ?custo) e
(?proposal http://semanticore.pucrs.br#duracao ?duracao) e
sum (?duracao, ?duracao, ?total) e
lessThan (?total, ?duracaoMax) e
sum (?custoMax, ?custoMax, ?total) e
lessThan (?custo, ?total))
postCondition:
(?request http://semanticore.pucrs.br#status "confirmaçãoEnviada")</td></tr>
</table>

Tabela A.2: Itens do objeto de conhecimento "KO-2"

<table>
<tr><th>Instância/Item</th><th colspan="2">Atributos da instância</th></tr>
<tr><td>descriptorKO-2/
DescriptorITEM</td><td colspan="2">- kOName: "KO-2"
- kOType: "domínio"
- description: "Conhecimento para o agendamento de consultas e tratamentos sem verificação do tratamento ou localização do paciente".
- status: "acabado"
- domain: "Agendamento de compromissos"
- author: "Ana Paula Lemke"
- creationDate: "10/12/2006"</td></tr>
<tr><td>goalKO-2/
GoalITEM</td><td colspan="2">- hasOntology: vide Figura 6.10.
- hasVerificationSentence:
("numClinicas" ">" "0")
- criterion:
criterionClass: "organizer.hostpots.SimpleGoalCriterion"
name: "SimpleGoalCriterion"
weight: "1"</td></tr>
<tr><td>sensorKO-2/
SensorITEM</td><td colspan="2">- name: "sensorKO-2"
- sensorClass: "semanticore.agent.sensorial.hotspots.OWLSensor"
- headerPattern:
(?id http://semanticore.pucrs.br#to agentID)</td></tr>
<tr><td rowspan="4">planKO-2/
ActionPlanITEM</td><td colspan="2">- name: "planKO-2"
- context:
name: "numClinicas"</td></tr>
<tr><td rowspan="3">- hasAction</td><td>name: "AdquirirPrescricao"
preCondition:
(?paciente rdf:type http://semanticore.pucrs.br#Paciente) e
(?paciente http://semanticore.pucrs.br#nome ?nome) e
(?medico rdf:type http://semanticore.pucrs.br#Medico) e
(?medico http://semanticore.pucrs.br#nome ?nomeM)
postCondition:
(?buscaClinica http://semanticore.pucrs.br#sts "AguardandoTrat") e
(?buscaClinica http://semanticore.pucrs.br#nome ?nome)</td></tr>
<tr><td>name: "SolicitarServicos"
(?buscaClinica http://semanticore.pucrs.br#sts "AguardandoTrat") e
(?t rdf:type http://semanticore.pucrs.br#Tratamento) e
(?t http://semanticore.pucrs.br#destinatario ?paciente) e
(?t http://semanticore.pucrs.br#palavrasChave ?palavras) e
(?paciente http://semanticore.pucrs.br#nome ?nomeP) e
equal (?nome, nomeP)
postCondition:
(?buscaClinica http://semanticore.pucrs.br#sts "SolEnviada")</td></tr>
<tr><td>name: "SelecionarClinicas"
preCondition:
(? buscaClinica http://semanticore.pucrs.br#sts " SolEnviada")
(?clinica rdf:type http://semanticore.pucrs.br#Clinica)
postCondition:
(? buscaClinica http://semanticore.pucrs.br#sts "acabado")</td></tr>
</table>

Tabela A.3: Itens do objeto de conhecimento "KO-3"

Instância/Item	Atributos da instância	
descriptorKO-3/ DescriptorITEM	- *kOName*: "KO-3" - *kOType*: "domínio" - *description*: "Conhecimento para o agendamento de consultas e tratamentos, considerando o plano de saúde do paciente e a sua localização". - *status*: "acabado" - *domain*: "Agendamento de compromissos" - *author*: "Ana Paula Lemke" - *creationDate*: "10/12/2006"	
goalKO-3/ GoalITEM	- *hasOntology*: vide Figura 6.9. - *hasVerificationSentence*: ("numClinicas" ">" "0") - *criterion*: *criterionClass*: "organizer.hostpots.SimpleGoalCriterion" *name*: "SimpleGoalCriterion" *weight*: "1"	
sensorKO-3/ SensorITEM	- *name*: "sensorKO-3" - *sensorClass*: "semanticore.agent.sensorial.hotspots.OWLSensor" - *headerPattern*: (?id http://semanticore.pucrs.br#to agentID)	
planKO-3/ ActionPlanITEM	- *name*: "planKO-3" - *context*: *name*: "numClinicas"	
	- *hasAction*	*name*: "AdquirirPrescricao" *preCondition*: (?paciente rdf:type http://semanticore.pucrs.br#Paciente) e (?paciente http://semanticore.pucrs.br#nome ?nome) e (?medico rdf:type http://semanticore.pucrs.br#Medico) e (?medico http://semanticore.pucrs.br#nome ?nomeM) e (?plano rdf:type http://semanticore.pucrs.br#PlanoSaude) e (?plano http://semanticore.pucrs.br#nome ?nomeP) e (?paciente http://semanticore.pucrs.br#possuiPlano ?plano) e (?localizacao rdf:type http://semanticore.pucrs.br#Localizacao) e (?localizacao http://semanticore.pucrs.br#latitude ?latitude) e (?localizacao http://semanticore.pucrs.br#longitude ?longitude) e (?paciente http://semanticore.pucrs.br#possuiEndereco ?localizacao) e (?clinicaX rdf:type http://semanticore.pucrs.br#Clinica) e (?clinicaX http://semanticore.pucrs.br#distanciaMax ?dist) *postCondition*: (?buscaServico http://semanticore.pucrs.br#sts "AguardandoTrat") e (?buscaServico http://semanticore.pucrs.br#nome ?nome)
		name: "RecuperarServicosPlano" *preCondition*: (?buscaServico http://semanticore.pucrs.br#sts "AguardandoTrat") e (?tratamento rdf:type http://semanticore.pucrs.br#Tratamento) e (?tratamento http://semanticore.pucrs.br#destinatario ?paciente) e (?paciente http://semanticore.pucrs.br#nome ?nomeP) e equal (?nome, nomeP) *postCondition*: (?buscaServico http://semanticore.pucrs.br#sts "SolEnviada")
		name: "RecuperarQualificacao" *preCondition*: (?buscaServico http://semanticore.pucrs.br#sts "SolEnviada") e (?clinica rdf:type http://semanticore.pucrs.br#Clinica) *postCondition*: (?buscaServico http://semanticore.pucrs.br#sts "QualiEnviada")

		name: “CalcularDistancia” *preCondition*: (?buscaServico http://semanticore.pucrs.br#sts “QualiEnviada”) e (?clinica rdf:type http://semanticore.pucrs.br#Clinica) e (?clinica http://semanticore.pucrs.br#possuiEndereco ?localizacao) *postCondition*: (?buscaServico http://semanticore.pucrs.br#sts “DistCalculada”)
		name: “SelecionarClinicas” *preCondition*: (?buscaServico http://semanticore.pucrs.br#sts “DistCalculada”) *postCondition*: (?buscaServico http://semanticore.pucrs.br#sts “acabado”)

Tabela A.4: Itens do objeto de conhecimento "KO-4"

Instância/Item	Atributos da instância	
descriptorKO-4/ DescriptorITEM	- *kOName*: "KO-4" - *kOType*: "domínio" - *description*: "Conhecimento para o agendamento de consultas e tratamentos por solicitação aos prestadores de serviço". - *status*: "acabado" - *domain*: "Agendamento de compromissos" - *author*: "Ana Paula Lemke" - *creationDate*: "10/12/2006"	
goalKO-4/ GoalITEM	- *hasOntology*: vide Figura 6.11. - *hasVerificationSentence*: ("numClinicas" ">" "0") - *criterion*: *criterionClass*: "organizer.hostpots.SimpleGoalCriterion" *name*: "SimpleGoalCriterion" *weight*: "1"	
sensorKO-4/ SensorITEM	- *name*: "sensorKO-4" - *sensorClass*: "semanticore.agent.sensorial.hotspots.OWLSensor" - *headerPattern*: (?id http://semanticore.pucrs.br#to agentID)	
planKO-4/ ActionPlanITEM	- *name*: "planKO-4" - *context*: *name*: "numClinicas"	
	- *hasAction*	*name*: "AdquirirPrescricao" *preCondition*: (?paciente rdf:type http://semanticore.pucrs.br#Paciente) e (?paciente http://semanticore.pucrs.br#nome ?nome) e (?medico rdf:type http://semanticore.pucrs.br#Medico) e (?medico http://semanticore.pucrs.br#nome ?nomeM) e (?plano rdf:type http://semanticore.pucrs.br#PlanoSaude) e (?plano http://semanticore.pucrs.br#nome ?nomeP) e (?paciente http://semanticore.pucrs.br#possuiPlano ?plano) e (?localizacao rdf:type http://semanticore.pucrs.br#Localizacao) e (?localizacao http://semanticore.pucrs.br#latitude ?latitude) e (?localizacao http://semanticore.pucrs.br#longitude ?longitude) e (?paciente http://semanticore.pucrs.br#possuiEndereco ?localizacao) e (?clinicaX rdf:type http://semanticore.pucrs.br#Clinica) e (?clinicaX http://semanticore.pucrs.br#distanciaMax ?dist) e *postCondition*: (?buscaClinica http://semanticore.pucrs.br#sts "AguardandoTrat") e (?buscaClinica http://semanticore.pucrs.br#nome ?nome)
		name: "SolicitarServicos" *preCondition*: (?buscaClinica http://semanticore.pucrs.br#sts "AguardandoTrat") e (?tratamento rdf:type http://semanticore.pucrs.br#Tratamento) e (?tratamento http://semanticore.pucrs.br#destinatario ?paciente) e (?tratamento http://semanticore.pucrs.br#palavrasChave ?palavras) e (?paciente http://semanticore.pucrs.br#nome ?nomeP) e equal (?nome, nomeP) *postCondition*: (?buscaClinica http://semanticore.pucrs.br#sts "SolEnviada")
		name: "RecuperarQualificacao" *preCondition*: (?buscaClinica http://semanticore.pucrs.br#sts "SolEnviada") e (?clinica rdf:type http://semanticore.pucrs.br#Clinica) *postCondition*: (?buscaClinica http://semanticore.pucrs.br#sts "QualiEnviada")
		name: "CalcularDistancia"

		preCondition: (?buscaClinica http://semanticore.pucrs.br#sts “QualiEnviada”) e (?clinica rdf:type http://semanticore.pucrs.br#Clinica) e (?clinica http://semanticore.pucrs.br#possuiEndereco ?localizacao) *postCondition*: (?buscaClinica http://semanticore.pucrs.br#sts “DistCalculada”)
		name: “SelecionarClinicas” *preCondition*: (?buscaClinica http://semanticore.pucrs.br#sts “DistCalculada”) e (?clinica rdf:type http://semanticore.pucrs.br#Clinica) e (?clinica http://semanticore.pucrs.br#planoAssociados ?plano) *postCondition*: (?buscaClinica http://semanticore.pucrs.br#sts “acabado”)